Int. J. Inf. Secur. (2007) 6:107–131
DOI 10.1007/s10207-007-0016-z

SPECIAL ISSUE PAPER

Transformational typing and unification for automatically correcting insecure programs

Boris Köpf · Heiko Mantel

Published online: 2 March 2007

Abstract Before starting a rigorous security analysis of a given software system, the most likely outcome is often already clear, namely that the system is not entirely secure. Modifying a program such that it passes the analysis is a difficult problem and usually left entirely to the programmer. In this article, we show that and how unification can be used to compute such program transformations. This opens a new perspective on the problem of correcting insecure programs. We also demonstrate that integrating our approach into an existing transforming type system can improve the precision of the analysis and the quality of the resulting programs.

Keywords Language-based security · Information flow control · Security type system · Unification

1 Introduction

1.1 Motivation

Restrictions on the permitted flow of information are often adequate for expressing security requirements like confidentiality or integrity. Information flow control goes beyond access control in that it not only controls which data a program accesses but also how the data is propagated within the program after a legitimate access. That is, the aim is to secure the process of computing, which includes providing adequate protection against unwanted information release and data corruption by accidentally flawed programs as well as by malicious code.

One approach to automatically analyzing the flow of information in concrete programs is the use of security type systems. Intuitively, the security type of a program captures how information can flow during its execution. For deriving the security type of a given program, one applies the rules of the type system. Given a security type, the rules can also be used to check if a program has this type. Usually, programs have a security type only if they cannot cause insecure information flow. That is, if type checking succeeds for some type then the given program is secure. Conversely, if type checking fails then the program might be insecure and should not be run.

When type checking fails, the task of correcting the program is usually left to the programmer. However, making a program secure can be intellectually challenging as well as tedious. Hence, it would be highly desirable to better support the programmer in this task.

We envision automated tool support on two levels:

1. for providing advice and suggestions to programmers on how a program could be interactively improved,
2. for automatically correcting some insecure programs or parts of programs without any need for interaction.

This is an extended version of the article [8] that appeared in Third International Workshop, FAST 2005, Revised Selected Papers, LNCS 3866, Springer-Verlag, 2006.
The second author gratefully acknowledges support by the German Research Foundation (DFG).

B. Köpf (✉)
Information Security, ETH Zurich, Zurich, Switzerland
e-mail: bkoepf@inf.ethz.ch

H. Mantel
Department of Computer Science,
RWTH Aachen University, Aachen, Germany
e-mail: mantel@cs.rwth-aachen.de

The long-term goal is a framework that incorporates both levels, but this is beyond the scope of the current article. Here, we focus solely on the second level, i.e., the fully automatic correction of some programs. For describing such program transformations, we employ transforming security type systems. The difference to non-transforming type systems is that if a given program is insecure then a transforming type system need not reject the program, but may modify it instead.

Obviously, one cannot allow an automatic transformation to modify programs in completely arbitrary ways. The corrected program should not only be secure, but also resemble the original program in a well-defined way.

1.2 Approach and contributions

We define the objectives of the program transformation with two equivalence relations: $\simeq$ and $\mathrel{\hat{=}}$. The first relationship captures in which sense a transformed program should resemble the original program. The second relationship captures the observational capabilities of the attacker by defining what is indistinguishable for him.

A transformation takes a program C as input and returns a transformed program C'. The transformation process is sound only if $C \simeq C'$ holds and if the attacker cannot distinguish between alternative executions of C' that occur depending on some secret. For instance, if the value of the guard B in a conditional $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ depends on a secret then the branches must be observationally equivalent for the attacker, i.e., $C_1 \mathrel{\hat{=}} C_2$, because, otherwise, the attacker might be able to deduce the secret. The partial equivalence relation (PER) model [13] even reduces the problem of making an entire program secure to the problem of making the program observationally equivalent to itself.

Our general framework for transformations is parametric in the relations $\simeq$ and $\mathrel{\hat{=}}$. On a technical level, a transformation makes use of meta-variables, substitutions, and unification. A given program is prepared for the transformation by inserting meta-variables in suitable positions. The relationship $\simeq$ provides a natural constraint on the placement of meta-variables and on the range of substitutions. The meta-variables allow us to reduce the problem of making alternative execution paths equivalent under $\mathrel{\hat{=}}$ to the problem of finding a suitable substitution for the inserted meta-variables. Unification is the means for computing such substitutions.

We investigate in detail an instance of our approach that eliminates differences in the timing behavior of alternative branches. Such differences could otherwise reveal to an attacker the actual execution path and, hence, the secrets on which the control flow depends. For this instance, we develop a calculus for inserting meta-variables and a transforming type system based on unification. We demonstrate that the problem of finding suitable substitutions in this instance can be reduced to a well-known unification problem: AC_1 unification with free constructors [7]. This reduction allows one to employ existing unification algorithms for computing unifiers, though this does not yet constitute an optimal solution. We propose an optimized unification algorithm that better exploits the specific structure of the unification problems originating during the transformation.

Rather than developing a type system completely from scratch, we build on an existing type system [14]. One benefit of this choice is that we can compare our approach in detail with the cross-copying technique [1] that is used in [14]. In comparison to [14], the main advantages of our transforming type system are:

Improved precision. Our type system accepts some secure programs that are rejected by the original type system, and it can correct some insecure programs that cannot be corrected by the original type system.
Better corrections. Our transformation returns programs that are faster and often substantially smaller in size.
Broader scope. Our transformation can be applied in the context of security policies with more than two levels.

Besides these technical advantages, unification yields a very natural perspective on the problem of making two programs observationally equivalent. However, we do not claim that using unification is the only way to achieve these technical advantages, or that it will solve all problems with repairing insecure programs. In fact, the described instance of our framework is the most convincing instance that we have found so far and indicates the likely class of transformations within our framework.

1.3 Related work

Analyzing the information flow in computing systems requires techniques that go beyond the analysis of safety and liveness properties. Rather than inspecting each of the traces in isolation, one must analyze the entirety of possible system behaviors [10]. In particular, it is not possible to dynamically enforce all information flow properties by techniques such as execution monitoring [15]. Static analysis, on the other hand, allows one to derive guarantees that cover all system traces.

Type-based approaches to analyzing the security of the information flow in concrete programs have received much attention in recent years. A good overview of the

research area language-based information flow security is given in [12]. Here, we focus on approaches that involve transformations for making programs secure. The focus of such transformations has been on the problem of making the branches of conditionals observationally equivalent for a given attacker. An early proposal, though not yet a transformation, can be found in the work by Volpano and Smith [16]. They investigate a simple multi-threaded programming language where they require entire conditionals to be executed atomically, ruling out differences in the duration of alternative execution paths. Moreover, they forbid assignments to variables that are observable for the attacker in order to avoid other differences in the observable behavior. However, the atomic execution of entire conditionals severely constrains parallelism. A less restrictive solution was proposed by Sabelfeld and Sands [14]. Their transforming type system sequentially composes each branch of a conditional with a program that simulates the timing behavior of the respective other branch. This cross-copying transformation ensures identical timing behavior of the branches in the transformed program. Like in [16], assignments to variables are forbidden if their values can be observed by the attacker. While the cross-copying technique does not require the atomic execution of conditionals, and hence appears more suitable for concurrent programs, it also causes a significant performance overhead by unnecessarily increasing the running time of conditionals (as elaborated in Sect. 6). Another problem is that the transformation can introduce non-termination into a program. This problem is addressed, and partially solved, in [11] by composing the cross-copied programs concurrently (rather than sequentially).

The cross-copying technique was originally proposed for a sequential language by Agat [1]. Recently, two further transformations have been proposed for the sequential setting. Barthe et al. [2] proposed a variant of the cross-copying technique that is based on a transaction mechanism. Each branch is wrapped in a transaction and then sequentially composed with the respective other branch. The transaction around the original branch is committed, while the cross-copied transaction is aborted. This technique is applicable to object-oriented languages with exceptions and methods calls. However, it is not clear if and how it could be lifted to a multi-threaded setting. Hedin and Sands [6] give a semantic security condition for alternative execution paths in the context of a sequential low-level language and informally sketch a transformation.

The approach introduced in the current article is suitable for a multi-threaded setting. In comparison to prior work, the performance overhead for a transformed program is minimized, and branches may contain assignments to variables that are observable by the attacker. A detailed, technical comparison is provided in Sect. 6.

1.4 Overview

Section 2 introduces the general framework for using unification in transformational typing. The framework is instantiated by defining concrete relations $\simeq$ and $\doteq$ in Sect. 3, and the instance is integrated into an existing transforming type system in Sect. 4. Possibilities for automating the transformation are investigated in Sect. 5. This includes the reduction to AC_1-unification as well as the proposal of a tailored unification algorithm. The merits of the proposed approach are discussed and put into perspective with related work in Sect. 6. Section 7 concludes the article. Proofs for all technical results can be found in the appendix.

2 A framework for making programs secure

2.1 Scope of the transformation

Assume the viewpoint of an attacker ζ who observes the program during execution. The security policy is that no information must flow from variables that store secret data to the variables that the attacker can observe during a program run. We use h to denote variables that store secrets and refer to such variables as *high variables*. Variables that ζ can observe are referred to as *low variables*, and we use l to denote a low variable.

We distinguish two classes of information leaks:

> *Intra-command leaks*. Information leakage occurs within a single command. For instance, after executing $l := h$ the terminal value of l equals the initial value of h.
> *Inter-command leaks*. Information leakage results from the interplay between different commands. For instance, after executing if h then $l := 1$ else $l := 0$, the terminal value of l equals the initial value of h.[1]

In this article, we focus on the removal of inter-command leaks. Generally, such leaks can occur if the control flow depends on a secret and if the alternative execution

[1] Intra-command leaks are also known as *explicit leaks* and inter-command leaks as *implicit leaks* [5]. Some authors further distinguish, e.g., *timing* and *termination leaks* [14].

paths yield different observations for the attacker. An inter-command leak involves some language primitive that conditionally affects the control flow like, e.g., an if-then-else statement, a while-loop, a conditional jump, a dynamic method dispatch, or an exception. For reducing the complexity, we focus on conditionals, i.e., if–then–else statements with high guards throughout the article.

2.2 Observational capabilities of the attacker

Inter-command leaks can be eliminated by making the branches of conditionals with high guards observationally equivalent. Such a transformation makes it impossible for a given attacker to deduce the value of the high guard as he cannot tell from his observations which branch was chosen. A prerequisite for applying this approach is a clear understanding of what the attacker ζ can observe during the execution of a program.

Common assumptions are, e.g., that the attacker can observe the values of low variables only at the end of a program run or that he can also observe the values of low variables in intermediate states. He might make further observations like, e.g., the time at which values of low variables are modified, whether a program run terminates, or how long a program run takes. Whether a given program is secure or not depends closely on the observational capabilities of the attacker.

Example 1 The program $P_1 = l := h$ copies the value of h into l and, hence, causes secret information to flow into l. This violates the security policy given that the attacker ζ can observe the values of low variables at least at the end of a program run. Similarly, the program $P_2 = \mathsf{if}\ h\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$ copies a Boolean value from h to l.

The program $P_3 = l := h; l := 0$ obeys the security policy given that ζ can only observe the values of low variables after termination. This is because the final value of l is 0, independent of the value of h. However, if ζ can observe the values of l in intermediate states during a run then the program is insecure.

The program $P_4 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ l := 1$ obeys the security policy under the assumption that ζ is not capable of observing the time at which the assignments to l occur and the duration of program execution. Otherwise, the program is insecure because the timing of the assignments and the duration of the execution both vary depending on the value of h.

The observational capabilities of an attacker can be expressed with equivalence relations. An equivalence relation $=_\zeta$ on states can be used to capture what the attacker ζ observes in an individual state: $s_1 =_\zeta s_2$ means that ζ makes the same observations in the state s_1 as in the state s_2. An equivalence relation $\doteq_\zeta$ on configurations (i.e., pairs of programs and states) can be used to capture that ζ's observations when program C_1 is run in state s_1 equal ζ's observations when C_2 is run in s_2.

Example 2 If ζ can observe only the value of the variable l then $s_1 =_\zeta s_2$ holds if and only if $s_1(l) = s_2(l)$, i.e., if the value of l in s_1 equals the value of l in s_2.

Consider the program $P_1 = l := h$ from Example 1 and states s_1 and s_2 such that $s_1(l) = s_2(l) = 0$, $s_1(h) = 0$, and $s_2(h) = 1$. Then $\langle P_1, s_1\rangle \neq_\zeta \langle P_1, s_2\rangle$ because $s_1'(l) = 0 \neq 1 = s_2'(l)$ where s_1' and s_2' is the state after running $l := h$ in s_1 and in s_2, respectively.

From notions of indistinguishablity of states and configurations one can derive a notion of indistinguishability of programs. Set $C_1 \doteq_\zeta C_2$ if and only if, for all observationally equivalent states s_1 and s_2 ($s_1 =_\zeta s_2$), the configurations (C_1, s_1) and (C_2, s_2) are observationally equivalent, i.e., $(C_1, s_1) \doteq_\zeta (C_2, s_2)$. The resulting relation on programs is only a PER, i.e., a transitive and symmetric relation that need not be reflexive. Not being observationally equivalent to itself means for a program C that running C in two indistinguishable states may lead to different observations. In this way, differences between the initial states are revealed, which lets ζ learn secret information. This view point is the key to capturing secure information flow in the PER model [13] in which a program is secure if and only if it is observationally equivalent to itself.

Example 3 Consider $P_1 = l := h$ and the attacker ζ from Example 2. In such a scenario, P_1 is not observationally equivalent to itself as there are two indistinguishable states s_1 and s_2 such that the configurations $\langle P_1, s_1\rangle$ and $\langle P_1, s_2\rangle$ are not observationally equivalent (as demonstrated in Example 2).

However, it not only depends on the observational capabilities of the attacker whether a given program should be considered secure or not. Another relevant factor is the context in which the program operates.

Example 4 Assume $P_4 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ l := 1$ from Example 1 runs concurrently with $P_5 = \mathsf{skip}; l := 0$ under a shared memory and a round robin scheduler. Then the final value of l is 0 given that the initial value of h is 0. Moreover, the final value of l is 1 given that the initial value of h is 1. This is illustrated below where (v_l, v_h) denotes the state s with $s(l) = v_l$ and $s(h) = v_h$:

$$
\begin{array}{ll}
\{\!|\langle P_4, P_5\rangle, (0,0)|\!\} & \{\!|\langle P_4, P_5\rangle, (0,1)|\!\} \\
\rightarrowtriangle \{\!|\langle l := 1, P_5\rangle, (0,0)|\!\} & \rightarrowtriangle \{\!|\langle (\mathsf{skip}; l := 1), P_5\rangle, (0,1)|\!\} \\
\rightarrowtriangle \{\!|\langle l := 1, l := 0\rangle, (0,0)|\!\} & \rightarrowtriangle \{\!|\langle (\mathsf{skip}; l := 1), l := 0\rangle, (0,1)|\!\} \\
\rightarrowtriangle \{\!|\langle l := 0\rangle, (1,0)|\!\} & \rightarrowtriangle \{\!|\langle l := 1, l := 0\rangle, (0,1)|\!\} \\
\rightarrowtriangle \{\!|\langle\rangle, (0,0)|\!\} & \rightarrowtriangle \{\!|\langle l := 1\rangle, (0,1)|\!\} \\
 & \rightarrowtriangle \{\!|\langle\rangle, (1,1)|\!\}
\end{array}
$$

That is, the final value of l equals the initial value of h and, hence, an attacker who can observe the final value of l is able to reconstruct the initial value of h.

We argued in Example 1 that P_4 is insecure if the attacker can observe the timing behavior of the program, and that it is secure, in a sequential context, if the attacker cannot observe the timing behavior. The interesting point illustrated by Example 4 is that the program becomes insecure in a multi-threaded context for the same attacker (i.e., one who is not able to observe the timing behavior). More generally, a program can be secure in a sequential setting for given observational capabilities of an attacker, while the same program is insecure in a concurrent setting for the same attacker. It is possible to take such aspects of the execution environment already into account when defining observational equivalence.

Example 5 Assume $\mathsf{skip}; l := 1 \not\doteq_\zeta l := 1$. For such a relation $\doteq_\zeta$, the program $P_4 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ l := 1$ from Example 4 is classified as insecure, because the branches of the conditional are not equivalent under $\doteq_\zeta$. This is a sensible classification of P_4 if programs under analysis shall run in parallel with other threads.

2.3 The transformation

We view the problem of making the branches of a conditional equivalent under a given relation $\doteq_\zeta$ as a unification problem on lifted programs. To lift a program, we insert meta-variables that can be substituted during the transformation. Suitable substitutions are computed by unification, for instance, when applying a transforming typing rule for conditionals of the following form:

$$
\frac{C_1 \hookrightarrow C_1' \quad C_2 \hookrightarrow C_2' \quad \sigma \in \mathcal{U}(\{C_1' \doteq_\zeta C_2'\})}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2'}
$$

That is, each branch is transformed into a secure program, resulting in C_1' and C_2', respectively. Unification is used to compute a unifier σ that makes C_1' and C_2' observationally equivalent under the relation $\doteq_\zeta$. The transformed program, $\mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2'$, is intuitively secure because each branch is secure (assuming secure programs with meta-variables cannot be made insecure by applying substitutions) and the branches are observationally equivalent. Since $\doteq_\zeta$ is reflexive in the PER model if restricted to secure programs and since unification is only applied on secure programs, we use the term *unification under an equational theory* in the following.

The typing rules for other language primitives simply propagate the transformations introduced when applying the rule for conditionals. For instance, a rule for sequential composition could have the following form:[2]

$$
\frac{C_1 \hookrightarrow C_1' \quad C_2 \hookrightarrow C_2'}{C_1; C_2 \hookrightarrow C_1'; C_2'}
$$

In summary, our approach proceeds in three steps:

1. lift a program by inserting meta-variables;
2. apply transforming typing rules to the lifted program;
3. eliminate all remaining meta-variables.

The approach is not only parametric in the equational theory under which branches are unified, but also in where meta-variables are placed and how they may be substituted. The latter two parameters determine, on the one hand, how similar a transformed program is to the original program and, on the other hand, limit the extent to which insecure programs can be corrected.

Example 6 If one decides to insert meta-variables between every two subcommands and to permit the substitution of meta-variables with arbitrary programs then lifting $P_2 = \mathsf{if}\ h\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$ results in

$$\mathsf{if}\ h\ \mathsf{then}\ (\alpha_1; l := 1; \alpha_2)\ \mathsf{else}\ (\alpha_3; l := 0; \alpha_4).$$

An example for a unifier of the branches is the substitution $\{\alpha_1 \backslash l := 0,\ \alpha_2 \backslash \epsilon, \alpha_3 \backslash \epsilon,\ \alpha_4 \backslash l := 1\}$. Note that this is a unifier under any equational theory as the instantiated program has syntactically identical branches.

If one restricts the range of substitutions to sequences of skip statements then a transformed program essentially is a slowed-down version of the original program. While this ensures that a transformed program more closely resembles the original program, it also makes it impossible to correct some programs like, e.g., P_2. However, $P_4 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ l := 1$, which is insecure in a multi-threaded setting (see Example 4), can still be corrected to $\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ (\mathsf{skip}; l := 1)$.

[2] Typing rules for other primitives that more actively contribute to the transformation are possible, but outside the scope of this paper. Here, the focus is on the elimination of inter-command leaks involving if-then-else statements.

If one relaxes the range of substitutions by allowing meta-variables to be substituted with functions from programs to programs then lifting P_2 might lead to

$$\mathsf{if}\ h\ \mathsf{then}\ \alpha_1(l := 1)\ \mathsf{else}\ \alpha_2(l := 0).$$

The two substitutions $\{\alpha_1 \backslash (\lambda x.\mathsf{skip}),\ \alpha_2 \backslash (\lambda x.\mathsf{skip})\}$ and $\{\alpha_1 \backslash (\lambda x.x),\ \alpha_2 \backslash (\lambda x.l := 1)\}$ are examples for unifiers of the branches. Applying them results in the programs $\mathsf{if}\ h\ \mathsf{then\ skip\ else\ skip}$ and $\mathsf{if}\ h\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 1$, respectively, which are both intuitively secure.

Generalizing the range of substitutions to functions creates more flexibility for program transformations, but it also results in more difficult unification problems where higher-order unification is needed.

2.4 Preservation of program behavior

When eliminating information leaks in a given program, one necessarily has to change the program's behavior in some way. In our approach such modifications are caused by the insertion and the subsequent instantiation of meta-variables. Whether one is able to correct a given insecure program depends largely on how much one is willing to modify the program's semantics. This trade-off is inherent in the problem of making a program secure and not a peculiarity of our approach.

Example 7 $P_4 = \mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ l := 1$ leaks information from high to low if the program executes in a concurrent setting (see Example 4). The leak can be eliminated by making the timing of the assignments in the two branches identical. Possible corrections are, e.g., $\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ (\mathsf{skip}; l := 1)$ and $\mathsf{if}\ h\ \mathsf{then}\ l := 1\ \mathit{else}\ l := 1$.

$P_2 = \mathsf{if}\ h\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$ leaks information from high to low (see Example 1). In order to eliminate this leak, more substantial changes to the program's behavior are needed. Possible corrections are $\mathsf{if}\ l'\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$, $\mathsf{if}\ h\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 1$, $\mathsf{if}\ h\ \mathsf{then}\ (l := 1; l := 0)\ \mathsf{else}\ (l := 1; l := 0)$, and $\mathsf{if}\ h\ \mathsf{then\ skip\ else\ skip}$. While the first three corrections cause quite significant changes, the fourth correction appears less problematic. Interestingly, this correction has similar effects to executing the original program under a dynamic information flow control that skips the execution of statements at run time if their execution would cause illegitimate information flow [5].

These examples just illustrate the wide spectrum of possible choices for defining in which sense a transformed program must be equivalent to the original program. Ultimately it depends on the application, how flexible one is in dealing with the trade-off between being able to correct more insecure programs and having transformed programs that closely resemble the original programs.

Hence, the question is not merely *whether* one is willing to change a given insecure program, but rather *how much* one is willing to let a fully automatic transformation change the program in order to eliminate information leaks. Specifying a range for substitutions and where meta-variables may be placed does not yet provide an adequate perspective for capturing how similar a transformed program will be to the original program. Capturing this by a notion of program equivalence appears more natural. Here, we assume that the desired relationship between a program and its transformation is given by an equivalence relation $\simeq$. That is, $C \simeq C'$ shall hold for the original program C and the transformed program C'. Alternatively, one could use simulation relations for this purpose as in [1,14], for instance.

The focus of this article will be on notions of program equivalence that permit differences in the timing behavior within individual threads, but no other changes. This makes corrections like the first one in Example 7 possible, but not the other corrections. While it appears difficult, if not impossible, to define program equivalences that permit the latter corrections and that are also generally acceptable, this does not rule out the possibility of defining a permissible program equivalence that is appropriate for a particular application or a particular program.

3 Instantiating the approach

We are now ready to illustrate how our approach can be instantiated. We introduce a simple programming language, a security policy, an observational equivalence relation, and a notion of program equivalence to be preserved under the transformation.

3.1 Programming language

We adopt a multi-threaded while language (MWL) from [14], which includes skip, assignments, conditionals, loops, and a command for dynamic thread creation. The set *Com* of *commands* is defined by

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(CV) \end{aligned}$$

where V is a command vector in $\boldsymbol{Com} = \bigcup_{n\in\mathbb{N}} Com^n$.

Expressions are variables, constants, or terms resulting from applying binary operators to expressions. A *state* is a mapping from variables in a given set *Var* to values in a given set *Val*. We use the judgment $\langle Exp, s\rangle \downarrow n$

for specifying that expression *Exp* evaluates to value *n* in state *s*. Expression evaluation is assumed to be total and to occur atomically. We say that expressions *Exp* and *Exp*′ are *equivalent* (denoted by $Exp \equiv Exp'$) if and only if they evaluate to identical values in each state, i.e., $\forall s \in S : \forall v \in Val : \langle Exp, s\rangle \downarrow v \Leftrightarrow \langle Exp', s\rangle \downarrow v$.

A *configuration* is a pair $\langle V, s\rangle$ where the vector V specifies the threads that are currently active and s defines the current state of the memory.

The operational semantics for MWL is formalized in Figs. 1 and 2. *Deterministic judgments* have the form $\langle C, s\rangle \rightarrowtriangle \langle W, t\rangle$ expressing that command C performs a computation step in state s, yielding a state t and a vector of commands W, which has length zero if C terminated, length one if it has neither terminated nor spawned any threads, and length greater than one if new threads were spawned. That is, a command vector of length n can be viewed as a *pool of n threads* that run concurrently. *Nondeterministic judgments* have the form $\langle V, s\rangle \rightarrowtriangle \langle V', t\rangle$, where V and V' are thread pools, expressing that some thread C_i in V performs a step in state s resulting in the state t and some thread pool W'. The global thread pool V' results then by replacing C_i with W'. For simplicity, we do not distinguish between commands and command vectors of length one in the notation and use the term *program* for referring to commands as well as to command vectors.

In the following, we adopt the naming conventions used above: s and t denote states, *Exp* denotes an expression, B denotes a Boolean expression, C denotes a command, and V and W denote command vectors.

$$\frac{\langle C_i, s\rangle \rightarrowtriangle \langle W', t\rangle}{\langle\langle C_0 \ldots C_{n-1}\rangle, s\rangle \rightarrowtriangle \langle\langle C_0 \ldots C_{i-1}\rangle W' \langle C_{i+1} \ldots C_{n-1}\rangle, t\rangle}$$

Fig. 1 Small-step nondeterministic semantics

$$\langle \mathsf{skip}, s\rangle \rightarrowtriangle \langle\langle\rangle, s\rangle \qquad \frac{\langle Exp, s\rangle \downarrow n}{\langle Id{:=}Exp, s\rangle \rightarrowtriangle \langle\langle\rangle, [Id = n]s\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrowtriangle \langle\langle\rangle, t\rangle}{\langle C_1; C_2, s\rangle \rightarrowtriangle \langle C_2, t\rangle} \qquad \frac{\langle C_1, s\rangle \rightarrowtriangle \langle\langle C_1'\rangle V, t\rangle}{\langle C_1; C_2, s\rangle \rightarrowtriangle \langle\langle C_1'; C_2\rangle V, t\rangle}$$

$$\langle \mathsf{fork}(CV), s\rangle \rightarrowtriangle \langle\langle C\rangle V, s\rangle$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \rightarrowtriangle \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \rightarrowtriangle \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \rightarrowtriangle \langle C; \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \rightarrowtriangle \langle\langle\rangle, s\rangle}$$

Fig. 2 Small-step deterministic semantics

3.2 Security policy and labelings

We assume a security policy that comprises two security domains, a *high level* and a *low level* where the requirement is that no information flows from *high* to *low*. This is the simplest policy for which the problem of information flow security can be investigated. Each program variable is associated with a security domain by means of a *labeling* $lab : Var \rightarrow \{low, high\}$. The intuition is that values of *low variables* can be observed by the attacker and, hence, should only be used to store public data. *High variables* are used for storing secret data and their contents are not observable for the attacker.

As mentioned before, h and l are used to denote high variables and low variables, respectively. An expression *Exp* has the security domain *low* (denoted by $Exp : low$) if all variables in *Exp* have domain *low* and, otherwise, has security domain *high* (denoted by $Exp : high$). The intuition is that values of expressions with domain *high* possibly depend on secrets while values of *low* expressions can only depend on public data.

3.3 Observational equivalence

We aim at securing programs against attackers who can observe, in a given state, the values of low variables and, in a program run, the values of low variables only in the terminal state. That is, our attackers can neither observe intermediate states in a run nor the duration of a run (i.e., the timing behavior is not observable). The programs are meant to be executed in a concurrent setting, in parallel with other threads, which implies that timing differences within alternative execution paths are potentially dangerous (see Example 4).

The concurrent setting can be taken into account in the definition of observational equivalence by pretending that the attacker were able to see the values of low variables at any point during a program run. Such an attacker can distinguish states s_1 and s_2 unless they are low equal (denoted by $s_1 =_L s_2$), i.e., if $\forall var \in Var : lab(var) = low \implies s_1(var) = s_2(var)$. Moreover, he cannot distinguish two program runs that have equal length and in which every two corresponding states are low equal. For capturing this intuition, Sabelfeld and Sands introduce the notion of a strong low bisimulation.

Definition 1 ([14]) The *strong low-bisimulation* $\cong_L$ is the union of all symmetric relations R on command vectors $V, V' \in \boldsymbol{Com}$ of equal size, i.e., $V = \langle C_1, \ldots, C_n\rangle$

and $V' = \langle C'_1, \ldots, C'_n \rangle$, such that

$$\begin{aligned}&\forall s, s', t \in S : \forall i \in \{1 \ldots n\} : \forall W \in \textbf{\textit{Com}}: \\ &[(V\, R\, V' \wedge s =_L s' \wedge \langle\!| C_i, s |\!\rangle \twoheadrightarrow \langle\!| W, t |\!\rangle) \\ &\quad \Rightarrow \exists W' \in \textbf{\textit{Com}}: \exists t' \in S : (\langle\!| C'_i, s' |\!\rangle \twoheadrightarrow \langle\!| W', t' |\!\rangle \\ &\qquad\qquad \wedge\, W\, R\, W' \wedge t =_L t')]\end{aligned}$$

The relation $\doteq_L \subseteq \textbf{\textit{Com}} \times \textbf{\textit{Com}}$ defined by the rules in Fig. 3 also captures this intuition. Moreover, this relationship provides a syntactic approximation of the strong bisimulation relation as programs that are related by $\doteq_L$ are also strongly bisimilar. It is not difficult to see that $\doteq_L$ is decidable if the equivalence of expressions can be decided. Interestingly, a similar result also holds for the strong low-bisimulation relation [4]. The relation $\doteq_L$ will serve us as observational equivalence in the following.

Theorem 1 (Adequacy of $\doteq_L$) *If $V \doteq_L V'$ is derivable then $V \approx_L V'$ holds.*

$$\frac{}{\mathsf{skip} \doteq_L \mathsf{skip}}\ [Skip] \qquad \frac{Id : high}{\mathsf{skip} \doteq_L Id{:=}Exp}\ [SHA_1]$$

$$\frac{Id : high}{Id{:=}Exp \doteq_L \mathsf{skip}}\ [SHA_2] \qquad \frac{Id : high \quad Id' : high}{Id{:=}Exp \doteq_L Id'{:=}Exp'}\ [HA]$$

$$\frac{Id : low \quad Exp : low \quad Exp' : low \quad Exp \equiv Exp'}{Id{:=}Exp \doteq_L Id{:=}Exp'}\ [LA]$$

$$\frac{C_1 \doteq_L C'_1 \quad C_2 \doteq_L C'_2}{C_1; C_2 \doteq_L C'_1; C'_2}\ [SComp]$$

$$\frac{C_1 \doteq_L C'_1, \ldots, C_n \doteq_L C'_n}{\langle C_1, \ldots, C_n \rangle \doteq_L \langle C'_1, \ldots, C'_n \rangle}\ [PComp]$$

$$\frac{C \doteq_L C' \quad V \doteq_L V'}{\mathsf{fork}(CV) \doteq_L \mathsf{fork}(C'V')}\ [Fork]$$

$$\frac{B, B' : low \quad B \equiv B' \quad C \doteq_L C'}{\mathsf{while}\ B\ \mathsf{do}\ C \doteq_L \mathsf{while}\ B'\ \mathsf{do}\ C'}\ [WL]$$

$$\frac{B, B' : low \quad B \equiv B' \quad C_1 \doteq_L C'_1 \quad C_2 \doteq_L C'_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2}\ [LIte]$$

$$\frac{B' : high \quad C_1 \doteq_L C'_1 \quad C_1 \doteq_L C'_2}{\mathsf{skip}; C_1 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2}\ [SHIte_1]$$

$$\frac{B : high \quad C_1 \doteq_L C'_1 \quad C_2 \doteq_L C'_1}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L \mathsf{skip}; C'_1}\ [SHIte_2]$$

$$\frac{Id : high \quad B' : high \quad C_1 \doteq_L C'_1 \quad C_1 \doteq_L C'_2}{Id{:=}Exp; C_1 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2}\ [HAHIte_1]$$

$$\frac{Id' : high \quad B : high \quad C_1 \doteq_L C'_1 \quad C_2 \doteq_L C'_1}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L Id'{:=}Exp'; C'_1}\ [HAHIte_2]$$

$$\frac{B, B' : high \quad C_1 \doteq_L C'_1 \quad C_1 \doteq_L C'_2 \quad C_1 \doteq_L C_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2}\ [HIte]$$

Fig. 3 A notion of observational equivalence

The proofs of this and all subsequent results are provided in Appendix A.

Remark 1 Note that it follows from Fig. 3 that $\doteq_L$ is a partial equivalence relation, i.e., it is transitive and symmetric, but not reflexive. For instance, $l := h$ is not $\doteq_L$-related to itself because the precondition of $[LA]$, the only rule in Fig. 3 applicable to assignments to low variables, rules out that high variables occur on the right hand side of the assignment. Nevertheless, $\doteq_L$ is an equivalence relation, even a congruence relation, if one restricts programs to the language *Slice* that we define, following [14], as the largest sub-language of *Com* without assignments of high expressions to low variables, assignments to high variables, and loops or conditionals having high guards. The sub-language *Slice* provides the context in which we will apply unification.

Similarly, $\approx_L$ is only a partial equivalence relation. For instance, the program $l := h$ is not strongly low bisimilar to itself. The argument is analogous to the one in Example 2. If one runs the program in states s_1 and s_2 with $s_1(l) = s_2(l) = 0$, $s_1(h) = 0$, and $s_2(h) = 1$ then one obtains states that are not low equal. Again, $\approx_L$ is an equivalence if restricted to programs in *Slice*. □

3.4 Program equivalence

We introduce an equivalence relation $\simeq$ to constrain the modifications caused by the transformation. Intuitively, this relationship requires a transformed program to be a slowed down version of the original program.

Definition 2 The *weak possibilistic bisimulation* $\simeq$ is the union of all symmetric relations R on command vectors such that whenever $V\, R\, V'$ then for all states s, t and all vectors W there is a vector W' such that

$$\begin{aligned}&\langle\!| V, s |\!\rangle \twoheadrightarrow \langle\!| W, t |\!\rangle \implies (\langle\!| V', s |\!\rangle \twoheadrightarrow^* \langle\!| W', t |\!\rangle \wedge WRW') \\ &\text{and } V = \langle\rangle \implies \langle\!| V', s |\!\rangle \twoheadrightarrow^* \langle\!| \langle\rangle, s |\!\rangle\ .\end{aligned}$$

The requirement that the transformed program must be $\simeq$-equivalent to the original program is stronger than the requirement in [14]. There, it is only required that the original program can simulate the transformed program. One advantage of our more restrictive choice is that a transformation cannot introduce nontermination.

4 Lifting a security type system

In this section we define a formal framework for transforming programs by inserting and instantiating meta-variables. Rather than developing an entirely new

formalism from scratch, we adapt an existing transforming type system from [14]. We show that any transformation within our framework is sound in the sense that the output is secure and the behavior of the original program is preserved in the sense of Definition 2.

4.1 Substitutions and liftings

We insert meta-variables from a set $\mathcal{V} = \{\alpha_1, \alpha_2, \dots\}$ into a program by sequential composition with its subterms. The extension of MWL with meta-variables is denoted by $\text{MWL}_\mathcal{V}$. The set $Com_\mathcal{V}$ of commands in $\text{MWL}_\mathcal{V}$ is defined by[3]

$$C ::= \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid C; X \mid X; C \\ \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(CV)\ ,$$

where placeholders X, Y range over $\mathcal{V}$. The set of all command vectors in $\text{MWL}_\mathcal{V}$ is $\boldsymbol{Com_\mathcal{V}} = \bigcup_{n\in\mathbb{N}}(Com_\mathcal{V})^n$. Note that the ground programs in $\text{MWL}_\mathcal{V}$ are exactly the programs in MWL. The operational semantics for ground programs remain unchanged, whereas programs with meta-variables are not meant to be executed. Along the same lines, we define the sets $Slice_\mathcal{V}$ and $\boldsymbol{Slice_\mathcal{V}}$, respectively, by inserting meta-variables into commands in *Slice*, and vectors thereof.

Meta-variables may be substituted with programs, meta-variables, or the special symbol ϵ that acts as the neutral element of the sequential composition operator (";"), i.e., $\epsilon; C = C$ and $C; \epsilon = C$.[4] When talking about programs in $Com_\mathcal{V}$ under a given substitution, we implicitly assume that these equations have been applied (from left to right) to eliminate the symbol ϵ from the program. Moreover, we view sequential composition as an associative operator and implicitly identify programs that differ only in the use of parentheses for sequential composition. That is, $C_1; (C_2; C_3)$ and $(C_1; C_2); C_3$ denote the same program.

Definition 3 A mapping $\sigma : \mathcal{V} \to (\{\epsilon\} \cup \mathcal{V} \cup Com_\mathcal{V})$ is a *substitution* if its *domain* $dom(\sigma) = \{\alpha \in \mathcal{V} \mid \sigma(\alpha) \neq \alpha\}$ is finite. The *range* of σ is defined as the set $ran(\sigma) = \sigma(dom(\sigma))$. A substitution mapping each meta-variable in a program V to $\{\epsilon\} \cup Com$ is a *ground substitution of* V. A substitution π mapping all meta-variables in V to ϵ is a *projection of* V. Given a program V in $\boldsymbol{Com}$, we call every program V' in $\boldsymbol{Com_\mathcal{V}}$ with $\pi V' = V$ a *lifting of* V.

[3] In the following, we overload notation by using C and V to denote commands and command vectors in $Com_\mathcal{V}$ (rather than in *Com*), respectively.

[4] Note that skip is not a neutral element of (";") as skip requires a computation step.

Example 8 The program

$$\mathsf{if}\ h\ \mathsf{then}\ (\alpha_1; \mathsf{skip}; \alpha_2; l := 1)\ \mathsf{else}\ (\alpha_3; l := 1)$$

is a lifting of $\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{skip}; l := 1)\ \mathsf{else}\ l := 1$. □

In the remainder of this article, we will focus on substitutions with a restricted range.

Definition 4 A substitution with range $\{\epsilon\} \cup Stut_\mathcal{V}$ is called *preserving*, where $Stut_\mathcal{V}$ is defined by

$$C ::= X \mid \mathsf{skip} \mid C_1; C_2$$

where the C_i range over $Stut_\mathcal{V}$.

The term *preserving* substitution is justified by the fact that such substitutions preserve a given program's semantics as specified in Definition 2.

Theorem 2 (Preservation of behavior)

1. *For all preserving substitutions* σ, ρ *that are ground for* $V \in \boldsymbol{Com_\mathcal{V}}$, *we have* $\sigma(V) \simeq \rho(V)$.
2. *For each lifting* V' *of a ground program* $V \in \boldsymbol{Com}$ *and each preserving substitution* σ *with* $\sigma(V')$ *ground, we have* $\sigma(V') \simeq V$.

4.2 Unification of programs

The problem of finding a substitution that makes the branches of conditionals with high guards observationally equivalent can be viewed as the problem of finding a unifier for the branches under the equational theory $\doteq_L$.[5] To this end, we lift the relation $\doteq_L \subseteq \boldsymbol{Com} \times \boldsymbol{Com}$ to a relation on $\boldsymbol{Com_\mathcal{V}}$ that we also denote by $\doteq_L$.

Definition 5 $V_1, V_2 \in \boldsymbol{Com_\mathcal{V}}$ are *observationally equivalent* ($V_1 \doteq_L V_2$) iff $\sigma V_1 \doteq_L \sigma V_2$ holds for each preserving substitution σ that is ground for V_1 and V_2.

Definition 6 A $\doteq_L$*-unification problem* Δ is a finite set of statements of the form $V_i \doteq_L^? V_i'$, i.e.,

$$\Delta = \{V_0 \doteq_L^? V_0', \dots, V_n \doteq_L^? V_n'\}$$

with $V_i, V_i' \in \boldsymbol{Com_\mathcal{V}}$ for all $i \in \{0, \dots, n\}$. A substitution σ is a *preserving unifier for* Δ if and only if σ is preserving and $\sigma V_i \doteq_L \sigma V_i'$ holds for each $i \in \{0, \dots, n\}$.

[5] Note that we apply unification only to programs in the sub-language $Slice_\mathcal{V}$, for which $\doteq_L$ constitutes a congruence relation (see Remark 1).

Fig. 4 A transforming security type system for programs with meta-variables

$$\frac{}{\mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}}\ [Skp] \qquad \frac{Id : high}{Id{:=}Exp \hookrightarrow Id{:=}Exp : \mathsf{skip}}\ [Ass_h] \qquad \frac{Id : low \qquad Exp : low}{Id{:=}Exp \hookrightarrow Id{:=}Exp : Id{:=}Exp}\ [Ass_l]$$

$$\frac{}{X \hookrightarrow X : X}\ [Var] \qquad \frac{C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : S_1; S_2}\ [Seq] \qquad \frac{B : low \qquad C \hookrightarrow C' : S}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ S}\ [Whl]$$

$$\frac{C_1 \hookrightarrow C_1' : S_1 \quad \ldots \quad C_n \hookrightarrow C_n' : S_n}{\langle C_1, \ldots, C_n\rangle \hookrightarrow \langle C_1', \ldots, C_n'\rangle : \langle S_1, \ldots, S_n\rangle}\ [Par] \qquad \frac{C_1 \hookrightarrow C_1' : S_1 \qquad V_2 \hookrightarrow V_2' : S_2}{\mathsf{fork}(C_1 V_2) \hookrightarrow \mathsf{fork}(C_1' V_2') : \mathsf{fork}(S_1 S_2)}\ [Frk]$$

$$\frac{B : low \qquad C_1 \hookrightarrow C_1' : S_1 \qquad C_2 \hookrightarrow C_2' : S_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2}\ [Cond_l]$$

$$\frac{B : high \quad C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2 \quad \sigma \in \mathcal{U}(\{S_1 \doteq_L^? S_2\})}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2' : \mathsf{skip}; \sigma S_1}\ [Cond_h]$$

A $\doteq_L$-unification problem is *solvable* if the set of preserving unifiers $\mathcal{U}(\Delta)$ for Δ is not empty.

4.3 A transforming type system

The transforming type system in Fig. 4 has been derived from the one in [14]. We use the judgment

$$V \hookrightarrow V' : S$$

for denoting that the MWL$_\mathcal{V}$-program V can be transformed into an MWL$_\mathcal{V}$-program V'. The intention is that V' has secure information flow and reflects the semantics of V as specified by Definition 2. The *slice* S is a program that is in the sub-language $Slice_\mathcal{V}$ and describes the timing behavior of V'.

The novelty over [14] is that our type system operates on $\boldsymbol{Com_\mathcal{V}}$ (rather than on ***Com***) and that the rule for high conditionals has been altered. In the original type system, a high conditional is transformed by sequentially composing each branch with the slice of the respective other branch. Instead of cross-copying slices, our rule instantiates the meta-variables occurring in the branches using preserving unifiers. The advantages of this modification are discussed in Sect. 6. Note that the rule $[Cond_h]$ does not mandate the choice of a specific preserving unifier of the branches. Nevertheless, we can prove that the type system meets our previously described intuition about the judgment $V \hookrightarrow V' : S$. To this end, we employ Sabelfeld and Sands's [13] strong security condition for defining what it means for a program to have secure information flow. Many other definitions are possible (see e.g., [12]).

Definition 7 A program $V \in$ ***Com*** *is strongly secure* if and only if $V \approx_L V$ holds. A program $V \in \boldsymbol{Com_\mathcal{V}}$ *is strongly secure* if and only if σV is strongly secure for each substitution σ that is preserving and ground for V.

In the following, we use $V \simeq V'$ (where $V, V' \in \boldsymbol{Com_\mathcal{V}}$) as abbreviation for: $\sigma V \simeq \sigma V'$ holds for every substitution σ that is preserving and ground for V and for V'.

Theorem 3 (Soundness of the type system) *If $V \hookrightarrow V' : S$ can be derived then*

1. *V' has secure information flow,*
2. *$V \simeq V'$ holds, and*
3. *$V' \approx_L S$ holds.*

The following corollary is an immediate consequence of Theorems 2 and 3. It shows that lifting a program and then applying the transforming type system preserves a program's behavior in the desired way.

Corollary 1 *If $V^* \hookrightarrow V' : S$ is derivable for some lifting $V^* \in \boldsymbol{Com_\mathcal{V}}$ of a program $V \in$ **Com** then V' has secure information flow and $V \simeq V'$.*

5 Automating the transformation

In Sect. 4, we have shown our type system to be sound for any choice of liftings and preserving unifiers in the applications of rule $[Cond_h]$. For automating the transformation, we have to define more concretely where meta-variables are inserted and how unifiers are determined.

5.1 Automatic insertion of meta-variables

When lifting a program, one is faced with a trade off: inserting meta-variables means to create possibilities for correcting the program, but it also increases the complexity of the unification problem. Within this spectrum our objective is to minimize the number of inserted meta-variables without losing the possibility of correcting the program. To this end, consider the sub-language $Pad_\mathcal{V}$, the extension of $Stut_\mathcal{V}$ with assignments to high variables, which is defined by the following grammar:

$$C ::= X \mid \mathsf{skip} \mid Id_h := Exp \mid C_1; C_2$$

Here X is a placeholder for meta-variables in $\mathcal{V}$, Id_h is a placeholder for program variables in *Var* with domain *high*, and C_1, C_2 are placeholders for commands in $Pad_\mathcal{V}$.

Observe that two programs C_1 and C_2 in $Pad_{\mathcal{V}}$ are related via $\doteq_L$ whenever they contain the same number of constants, i.e., skips and assignments to high variables (denoted as $const(C_1) = const(C_2)$), and the same number of occurrences of each meta-variable α (denoted by $|C_1|_\alpha = |C_2|_\alpha$). The positioning of assignments and meta-variables within the programs, however, is irrelevant.

Lemma 1 *For two commands C_1 and C_2 in $Pad_{\mathcal{V}}$ we have $C_1 \doteq_L C_2$ if and only if $const(C_1) = const(C_2)$ and $\forall \alpha \in \mathcal{V}: |C_1|_\alpha = |C_2|_\alpha$.*

Moreover, observe that inserting one meta-variable next to another does not create new possibilities for correcting a program. This, together with Lemma 1, implies that inserting one meta-variable into every subprogram within $Pad_{\mathcal{V}}$ is sufficient for allowing every possible correction.

We use this insight to define the language $Mgl_{\mathcal{V}}$ of *most general liftings*. It is the sub-language of $Com_{\mathcal{V}}$ that contains all liftings of programs in *Com* in which the rightmost subterm of every sub-program in $Pad_{\mathcal{V}}$ is a (unique) meta-variable. A technical side-effect of the definition of $Mgl_{\mathcal{V}}$ is the fact that it greatly simplifies inductive proofs.

Definition 8 The language $Mgl_{\mathcal{V}}$ is the subset of commands given by the following grammar, in which every meta-variable occurs at most once.

$$L ::= P \mid P; Id_l := Exp; L \mid P; \mathsf{if}\ B\ \mathsf{then}\ L_1\ \mathsf{else}\ L_2; L \mid P; \mathsf{while}\ B\ \mathsf{do}\ L_1; L \mid P; \mathsf{fork}(L_1 V); L$$

Here Id_l is a placeholder for program variables in *Var* with domain *low*, L, L_1, L_2 are placeholders for commands in $Mgl_{\mathcal{V}}$, V is a placeholder for a command vector in $\boldsymbol{Mgl_{\mathcal{V}}}$, and P is a placeholder for a command of the form X or of the form $C; X$ with $C \in Pad_{\mathcal{V}}$.

The liftings in $\boldsymbol{Mgl_{\mathcal{V}}}$ are most general in the sense that if two programs can be made observationally equivalent for some lifting then they can be made equivalent for any lifting chosen from $\boldsymbol{Mgl_{\mathcal{V}}}$.

Lemma 2 *Let $V_i \in \boldsymbol{Com}$, with liftings $V_i' \in \boldsymbol{Com_{\mathcal{V}}}$ and $V_i^* \in \boldsymbol{Mgl_{\mathcal{V}}}$ for $i = 1, 2$. Suppose V_1^* (V_2^*) shares no meta-variables with V_1', V_2', and V_2^* (V_1', V_2', and V_1^*). Then we have*

$$\mathcal{U}(\{V_1' \doteq_L^? V_2'\}) \neq \emptyset \text{ implies } \mathcal{U}(\{V_1^* \doteq_L^? V_2^*\}) \neq \emptyset$$

The calculus $\rightharpoonup$: $\boldsymbol{Com} \to \boldsymbol{Com_{\mathcal{V}}}$ turns the choice of a lifting in $\boldsymbol{Mgl_{\mathcal{V}}}$ into an algorithm. The mapping is defined inductively: a fresh meta-variable is sequentially composed to the right hand side of each subprogram. Another fresh meta-variable is sequentially composed to the left hand side of each assignment to a low variable, fork, while loop, or conditional. A lifting of a sequentially composed program is computed by sequentially composing the liftings of the subprograms while removing the terminal variable of the left program. The full calculus is given in Fig. 5.

$$\frac{X \text{ fresh}}{\mathsf{skip} \rightharpoonup \mathsf{skip}; X} \qquad \frac{Id : high \quad X \text{ fresh}}{Id{:=}Exp \rightharpoonup Id{:=}Exp; X}$$

$$\frac{Id : low \quad X, Y \text{ fresh}}{Id{:=}Exp \rightharpoonup X; Id{:=}Exp; Y} \qquad \frac{C_1 \rightharpoonup C_1'; X \quad C_2 \rightharpoonup C_2'}{C_1; C_2 \rightharpoonup C_1'; C_2'}$$

$$\frac{C_1 \rightharpoonup C_1', \ldots, C_n \rightharpoonup C_n'}{\langle C_1, \ldots, C_n \rangle \rightharpoonup \langle C_1', \ldots, C_n' \rangle}$$

$$\frac{C_1 \rightharpoonup C_1' \quad V_2 \rightharpoonup V_2' \quad X, Y \text{ fresh}}{\mathsf{fork}(C_1 V_2) \rightharpoonup X; (\mathsf{fork}(C_1' V_2')); Y}$$

$$\frac{C \rightharpoonup C' \quad X, Y \text{ fresh}}{\mathsf{while}\ B\ \mathsf{do}\ C \rightharpoonup X; (\mathsf{while}\ B\ \mathsf{do}\ C'); Y}$$

$$\frac{C_1 \rightharpoonup C_1' \quad C_2 \rightharpoonup C_2' \quad X, Y \text{ fresh}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \rightharpoonup X; (\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'); Y}$$

Fig. 5 A calculus for computing most general liftings

Example 9 Applying the lifting calculus to the program $C = h_1 := 1; h_2 := 1; l := 0; h_3 := 2$ results in the lifted program $\overline{C} = h_1 := 1; h_2 := 1; \alpha_1; l := 0; h_3 := 2; \alpha_2$. The program $\overline{C}$ contains substantially fewer meta-variables than the program obtained by naive insertion of meta-variables between every two subprograms. However, it still allows for every possible correction, as we show in the subsequent lemma.

The liftings computed by $\rightharpoonup$ are most general in the sense of Lemma 2.

Lemma 3 *Let $V \in \boldsymbol{Com}$ and $\overline{V} \in \boldsymbol{Com_{\mathcal{V}}}$. If $V \rightharpoonup \overline{V}$ can be derived, then*

1. *$\overline{V}$ is a lifting of V and*
2. *$\overline{V} \in \boldsymbol{Mgl_{\mathcal{V}}}$.*

Putting Lemmas 2 and 3 together, we can prove the *completeness* of the calculus $\rightharpoonup$: if two programs can be made observationally equivalent for some choice of liftings, then they can also be made equivalent if we restrict ourselves to the liftings computed by $\rightharpoonup$.

Theorem 4 *Let $V_i \in \boldsymbol{Com}$ with liftings $V_i', \overline{V_i} \in \boldsymbol{Com_{\mathcal{V}}}$ for $i = 1, 2$. Suppose $\overline{V_1}$ ($\overline{V_2}$) shares no meta-variables*

with V_1', V_2', and $\overline{V_2}$ (V_1', V_2', and $\overline{V_1}$). If $V_1 \rightharpoonup \overline{V_1}$ and $V_2 \rightharpoonup \overline{V_2}$ can be derived, then

$\mathcal{U}(\{V_1' \doteq_L^? V_2'\}) \neq \emptyset$ *implies* $\mathcal{U}(\{\overline{V_1} \doteq_L^? \overline{V_2}\}) \neq \emptyset$.

Theorem 4 shows that the lifting calculus reduces the number of inserted meta-variables without losing solutions. However, the theorem has not yet the form that we will need for proving the completeness of the entire automation (see Sect. 5.3) because the preconditions of the theorem do not hold if one chooses $V_1' = V_2'$ (disjointness of meta-variables is violated for non-ground programs). This is why we need another, slightly different completeness result for the lifting calculus.

Theorem 5 *Let $V', \overline{V} \in \boldsymbol{Com_\mathcal{V}}$ be liftings of $V \in \boldsymbol{Com}$. Suppose $\overline{V}$ shares no meta-variables with V' and $V \rightharpoonup \overline{V}$ can be derived. Then*

$\mathcal{U}(\{V' \doteq_L^? V'\}) \neq \emptyset$ *implies* $\mathcal{U}(\{\overline{V} \doteq_L^? \overline{V}\}) \neq \emptyset$.

Theorem 5 shows that if a program can be repaired for some lifting (recall that if a program is $\doteq_L$-equivalent to itself, then it is also secure) then it can also be repaired if we restrict ourselves to the lifting computed by $\rightharpoonup$.

5.2 Automating unification

Integrating standard unification algorithms Standard algorithms for the unification modulo an associative and commutative operator with neutral element and constants (see, e.g., [3] for background information on AC_1 unification) build on a characterization of equality that is equivalent to the one in Lemma 1. This correspondence allows one to employ existing algorithms for AC_1-unification problems with constants and free function symbols (like, e.g., the one in [7]) to the unification problems that arise when applying the rule for conditionals and then to filter the output such that only preserving substitutions remain.[6]

On most general unifiers When applying our transformational type system to programs with nested high conditionals, the rule $[Cond_h]$ is applied iteratively. When choosing a unifier of the branches, care must be taken in order not to rule out possible corrections in future applications of the rule $[Cond_h]$. A natural way to avoid this pitfall is to use a *most general unifier*, i.e., a unifier that can be specialized to every other unifier. Unfortunately, AC_1-unification problems with constants do, in general, not allow for most general unifiers [3]. This result carries over to the theory $\doteq_L$.

Example 10 The substitutions $\eta_1 = \{\alpha_1 \backslash \epsilon, \alpha_2 \backslash \mathsf{skip}\}$ and $\eta_2 = \{\alpha_1 \backslash \mathsf{skip}, \alpha_2 \backslash \epsilon\}$ both solve the unification problem $\Delta = \{\alpha_1; \alpha_2 \doteq_L^? \mathsf{skip}\}$. In fact, every possible solution of Δ must be ground for $\alpha_1; \alpha_2$, so there can be no unifier σ with substitutions ρ_1 and ρ_2 such that $\eta_1 = \rho_1 \circ \sigma$ and $\eta_2 = \rho_2 \circ \sigma$ hold. In other words, no most general unifier exists for Δ.

As in AC_1-unification problems with constants, the role of most general unifiers for $\doteq_L$ can be replaced by the more general notion of a *complete set of unifiers*. For a given unification problem Δ, a complete set of unifiers is a set $\Sigma \subseteq \mathcal{U}(\Delta)$ such that for every unifier $\sigma \in \mathcal{U}(\Delta)$, there is a substitution ρ and a $\eta \in \Sigma$ such that $\sigma = \rho\eta$. Minimal complete sets of unifiers can be computed and used for transformational typing. To avoid backtracking in the search, such a type system could return an entire set of transformed commands. Typing a high conditional then amounts to computing a complete set of unifiers for each combination of a command from the set returned for the then-branch with a command returned for the else-branch. Each unifier is then applied to the respective pair, resulting in a set of possible transformations for the high conditional. The transformational typing process succeeds, if the set of possible transformations of a program is not empty. Otherwise, the program is rejected as incorrectable. Fortunately, we can avoid this explosion in complexity by using a more problem-tailored unification algorithm.

A problem-tailored solution We next present a unification algorithm that makes use of our additional information on the positions of the inserted meta-variables and the limited range of preserving substitutions. Recall that we operate on programs in $Slice_\mathcal{V}$, i.e., on programs without assignments to high variables, without assignments of high expressions to low variables, and without loops or conditionals having high guards.

The unification algorithm in Fig. 6 is given in form of a calculus for judgments of the form $C_1 \doteq_L^? C_2 :: \eta$, meaning that η is a preserving unifier of the commands C_1 and C_2. The operative intuition behind the algorithm is to scan two program terms from left to right and distinguish two cases: if both leftmost subcommands are free constructors, (low assignments, loops, conditionals and forks) they are compared and, if they agree, unification is recursively applied to pairs of corresponding subprograms and the residual programs (see rules $[Asg]$,

[6] For the reader familiar with AC_1 unification: In the language $Stut_\mathcal{V}$ one views ϵ as the neutral element, skip as the constant, and ; as the operator. Other language constructs, i.e., assignments, conditionals, loops, forks, and ; (outside the language $Stut_\mathcal{V}$) must be treated as free constructors.

Fig. 6 Unification calculus

$$\frac{Id : low \quad Exp_1 \equiv Exp_2}{Id{:=}Exp_1 \mathrel{\hat{=}}^?_L Id{:=}Exp_2 :: \emptyset}\ [Asg] \qquad \frac{C \in Stut_{\mathcal{V}} \cup \{\epsilon\}}{X \mathrel{\hat{=}}^?_L C :: \{X \backslash C\}}\ [Var_1] \qquad \frac{C \in Stut_{\mathcal{V}} \cup \{\epsilon\}}{C \mathrel{\hat{=}}^?_L X :: \{X \backslash C\}}\ [Var_2]$$

$$\frac{C_1 \mathrel{\hat{=}}^?_L C_2 :: \eta \quad C_1, C_2 \in Stut_{\mathcal{V}}}{X; C_1 \mathrel{\hat{=}}^?_L C_2 :: \eta[X \backslash \epsilon]}\ [Seq_1] \qquad \frac{C_1 \mathrel{\hat{=}}^?_L C_2 :: \eta \quad C_1, C_2 \in Stut_{\mathcal{V}}}{C_1 \mathrel{\hat{=}}^?_L X; C_2 :: \eta[X \backslash \epsilon]}\ [Seq_1']$$

$$\frac{C_1 \mathrel{\hat{=}}^?_L C_2 :: \eta \quad C_1, C_2 \in Stut_{\mathcal{V}}}{\mathsf{skip}; C_1 \mathrel{\hat{=}}^?_L \mathsf{skip}; C_2 :: \eta}\ [Seq_2] \qquad \frac{C_1 \mathrel{\hat{=}}^?_L C_1' :: \eta_1 \quad C_2 \mathrel{\hat{=}}^?_L C_2' :: \eta_2 \quad C_1, C_1' \in NSeq_{\mathcal{V}}}{C_1; C_2 \mathrel{\hat{=}}^?_L C_1'; C_2' :: \eta_1 \cup \eta_2}\ [Seq_3]$$

$$\frac{C_1 \mathrel{\hat{=}}^?_L C_1' :: \eta_1 \quad C_2 \mathrel{\hat{=}}^?_L C_2' :: \eta_2 \quad C_1, C_1' \in Stut_{\mathcal{V}} \cup \{\epsilon\} \quad C_2, C_2' \in NStut_{\mathcal{V}}}{C_1; C_2 \mathrel{\hat{=}}^?_L C_1'; C_2' :: \eta_1 \cup \eta_2}\ [Seq_4]$$

$$\frac{C \mathrel{\hat{=}}^?_L C' :: \eta_1 \quad V \mathrel{\hat{=}}^?_L V' :: \eta_2}{\mathsf{fork}(CV) \mathrel{\hat{=}}^?_L \mathsf{fork}(C'V') :: \eta_1 \cup \eta_2}\ [Frk] \qquad \frac{C_1 \mathrel{\hat{=}}^?_L C_1' :: \eta_1, \ldots, C_n \mathrel{\hat{=}}^?_L C_n' :: \eta_n}{\langle C_1, \ldots, C_n \rangle \mathrel{\hat{=}}^?_L \langle C_1', \ldots, C_n' \rangle :: \bigcup_{i=1}^n \eta_i}\ [Par]$$

$$\frac{C_1 \mathrel{\hat{=}}^?_L C_1' :: \eta_1 \quad C_2 \mathrel{\hat{=}}^?_L C_2' :: \eta_2 \quad B_1, B_2 : low \quad B_1 \equiv B_2}{\mathsf{if}\ B_1\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mathrel{\hat{=}}^?_L \mathsf{if}\ B_2\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' :: \eta_1 \cup \eta_2}\ [Ite] \qquad \frac{C_1 \mathrel{\hat{=}}^?_L C_2 :: \eta \quad B_1, B_2 : low \quad B_1 \equiv B_2}{\mathsf{while}\ B_1\ \mathsf{do}\ C_1 \mathrel{\hat{=}}^?_L \mathsf{while}\ B_2\ \mathsf{do}\ C_2 :: \eta}\ [Whl]$$

[*Whl*], [*Ite*], and [*Frk*]). If one leftmost subcommand is skip, both programs are decomposed into their maximal initial subprograms in $Stut_{\mathcal{V}}$ and the remaining program (see rule [Seq_4]).[7] The rule [Seq_3] can then be applied to the remainders, and separates the initial free constructors from the programs that are sequentially composed to their right hand side. Recursive decomposition eventually leads to unification problems on $Stut_{\mathcal{V}}$. The rule [Seq_2] removes initial skips, while the rules [Var_1], [Var_2], [Seq_1], and [Seq_1'] govern how basic unifiers are constructed. Note that the unifiers obtained from recursive application of the algorithm to sub-programs are combined by set union, which is admissible if the meta-variables in all subprograms are disjoint.

Lemma 4 *Let $V_1, V_2 \in \boldsymbol{Slice_{\mathcal{V}}}$ where no meta-variable occurs more than once in (V_1, V_2). If $V_1 \mathrel{\hat{=}}^?_L V_2 :: \eta$, then*

1. *$\eta \in \mathcal{U}(\{V_1 \mathrel{\hat{=}}^?_L V_2\})$, and*
2. *η is idempotent, and*
3. *$dom(\eta) \cup var(ran(\eta)) \subseteq var(V_1) \cup var(V_2)$, and*
4. *$V_1, V_2 \in \boldsymbol{Mgl_{\mathcal{V}}}$ implies $\eta V_1, \eta V_2 \in \boldsymbol{Mgl_{\mathcal{V}}}$,*

where $var(\cdot)$ returns the set of meta-variables occurring in a set of commands in $\boldsymbol{Com_{\mathcal{V}}}$.

As Property 2 of Lemma 4 shows, the unifiers η computed by our calculus are idempotent, i.e., $\eta \circ \eta = \eta$ holds. Together with the assumption that every meta-variable occurs at most once in (V_1, V_2), Property 3 is the reason why unifiers may be combined using set union. Property 4 of Lemma 4 is a substitute for the existence of most general unifiers as, in combination with Lemma 2, it implies that we do not lose the possibility for subsequent corrections by applying the unifiers computed with our calculus. This fact allows us to prove the completeness of our transformational approach in the following section.

[7] Formally, we define the language $NSeq_{\mathcal{V}}$ of commands in $Slice_{\mathcal{V}} \setminus \{\mathsf{skip}\}$ without sequential composition as a top-level operator. The language $NStut_{\mathcal{V}}$ is the set of commands in $Slice_{\mathcal{V}}$ given by the grammar $C ::= C_1; C_2$, where $C_1 \in NSeq_{\mathcal{V}}$ and $C_2 \in Slice_{\mathcal{V}}$.

5.3 Completeness

In the following we show the completeness of our approach, in the sense that every program that can be repaired by a process of inserting skip commands at arbitrary positions in the program can also be repaired by our method. In other words, by first applying the lifting calculus from Fig. 5 and then applying the transforming type system of Fig. 4, where unification is instantiated with the algorithm in Fig. 6, we do not lose any possible corrections. In particular, our approach is complete with respect to applying the transformational type system to arbitrary liftings and instantiating it with an unification algorithm of choice.

Theorem 6 (Completeness) *Let $V \in \boldsymbol{Com}$, $\overline{V}, W \in \boldsymbol{Com_{\mathcal{V}}}$, W be a lifting of V, and $V \rightharpoonup \overline{V}$.*

1. *If there is a preserving substitution σ with $\sigma W \mathrel{\hat{=}}_L \sigma W$, then $\overline{V} \hookrightarrow' V' : S$ for some $V', S \in \boldsymbol{Com_{\mathcal{V}}}$.*
2. *If $W \hookrightarrow W' : S$ for some $W', S \in \boldsymbol{Com_{\mathcal{V}}}$ then $\overline{V} \hookrightarrow' V' : S'$ for some $V', S' \in \boldsymbol{Com_{\mathcal{V}}}$.*

Here, the judgment $V \hookrightarrow' V' : S$ denotes a successful transformation of V to V' by the transformational type system, where the precondition $\sigma \in \mathcal{U}(\{S_1 \mathrel{\hat{=}}^?_L S_2\})$ is replaced by $S_1 \mathrel{\hat{=}}^?_L S_2 :: \sigma$ in rule [$Cond_h$].

6 Discussion

Sections 3, 4 and 5 made our novel approach to transformational typing concrete in the context of a multi-threaded programming language. In the following, we show that this instance compares favorably with the cross-copying technique. As outlined in Sect. 1.3, there are several variants of this technique, some of them being suitable for concurrent languages. In our comparison, we focus on the version of cross-copying for a concurrent language as proposed in [14]. The later version

in [11] constitutes a slight improvement regarding the introduction of non-termination, but this problem does not even occur in our setting. More recent proposals (like, e.g., [2]) aim at sequential languages, and it is not clear how they could be adapted to a concurrent setting.

6.1 Comparison

The type system introduced in Sect. 4 is capable of analyzing programs where assignments to low variables appear in the branches of conditionals with high guards, which is not possible with the type system in [14].

Example 11 If one lifts

$$C = \mathsf{if}\ h_1\ \mathsf{then}\ (h_2 := Exp_1; l := Exp_2)\ \mathsf{else}\ (l := Exp_2),$$

where $Exp_2 : low$, using our lifting calculus, applies our transforming type system, and finally removes all remaining meta-variables by applying a projection then this results in

$$\mathsf{if}\ h_1\ \mathsf{then}\ (h_2 := Exp_1; l := Exp_2)\ \mathsf{else}\ (\mathsf{skip}; l := Exp_2),$$

a program that is strongly secure and also weakly bisimilar to C. Note that the program C cannot be repaired by applying the type system from [14] as assignments to low variables occur in the branches.

Another advantage of our unification-based approach in comparison to the cross-copying technique is that the resulting programs are faster and smaller in size.

Example 12 The program

$$\mathsf{if}\ h\ \mathsf{then}\ (h_1 := Exp_1)\ \mathsf{else}\ (h_2 := Exp_2)$$

is returned unmodified by our type system, while the type system from [14] transforms it into the bigger program

$$\mathsf{if}\ h\ \mathsf{then}\ (h_1 := Exp_1; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; h_2 := Exp_2).$$

If one applies this type system a second time, one obtains an even bigger program, namely

$$\begin{array}{l}\mathsf{if}\ h\ \mathsf{then}\ \ (h_1 := Exp_1; \mathsf{skip}; \mathsf{skip}; \mathsf{skip})\\ \quad\mathsf{else}\ \ (\mathsf{skip}; \mathsf{skip}; \mathsf{skip}; h_2 := Exp_2).\end{array}$$

In contrast, our type system realizes a transformation that is idempotent, i.e., the program resulting from the transformation remains unmodified under a second application of the transformation. This property turns out to be helpful in the context of multi-level security policies (see Sect. 6.2).

The chosen instantiation of our approach preserves the program behavior in the sense of a weak bisimulation. Naturally, one can correct more programs if one is willing to relax this relationship between input and output of the transformation. For this reason, there are also some programs that cannot be corrected with our type system although they can be corrected with the type system in [14] (which assumes a weaker relationship between input and output).

Example 13 $\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{while}\ l\ \mathsf{do}\ (h_1 := Exp))\ \mathsf{else}\ (h_2 := 1)$ is rejected by our type system. The type system in [14] transforms it into the strongly secure program

$$\begin{array}{l}\mathsf{if}\ h\ \mathsf{then}\ (\mathsf{while}\ l\ \mathsf{do}\ (h_1 := Exp); \mathsf{skip})\\ \quad\mathsf{else}\ \ (\mathsf{while}\ l\ \mathsf{do}\ (\mathsf{skip}); h_2 := 1).\end{array}$$

Note that this program is not weakly bisimilar to the original program as the cross-copying of the while loop introduces possible non-termination.

If one wishes to permit such transformations, one could, for instance, choose a simulation instead of the weak bisimulation in a variant of our approach. This would allow for an extended range of substitutions beyond $Stut_{\mathcal{V}}$. For instance, for correcting the program in Example 13, one would need to instantiate a meta-variable with a while loop. We are confident that, in such a setting, using our approach would even further broaden the scope of corrections while retaining the advantage of transformed programs that are comparably small and fast. It is not clear to us yet, however, how the concurrent version of cross-copying from [11] could be simulated in our approach.

6.2 Multi-level security policies

Non-transforming security type systems for the two-level security policy can even be used to analyze programs under a policy with more domains. To this end, one performs multiple type checks where each pass ensures that no illegitimate information flow can occur into a designated domain. For instance, consider a three-domain policy with domains $\mathcal{D} = \{top, left, right\}$, where information may only flow from *left* and from *right* to *top*. To analyze a program under this policy, one considers all variables with label *top* and *left* as if labeled *high* in a first type check (ensuring that there is no illegitimate information flow to *right*) and, in a second type check, considers all variables with label *top* and *right* as if labeled *high*. There is no need for a type check from the perspective of *top* as all information may flow to *top*. When adopting this approach for transforming type systems, one must take into account that the guarantees established by the type check for one domain might not be preserved under the modifications caused by the transformation for another domain. Therefore,

one needs to iterate the process until a fixpoint is reached for all security domains.[8]

Example 14 For the three-level policy from above, the program $\mathsf{if}\ t\ \mathsf{then}\ (t := t';\ r := r'; l := l')\ \mathsf{else}\ (r := r'; l := l')$ (assuming $t, t' : top$, $r, r' : right$ and $l, l' : left$) is lifted to $\mathsf{if}\ t\ \mathsf{then}\ (t := t'; r := r'; \alpha_1; l := l'; \alpha_2)\ \mathsf{else}\ (r := r'; \alpha_3; l := l'; \alpha_4)$ and transformed into

$$\begin{array}{l}\mathsf{if}\ t\ \mathsf{then}\ \ (t := t'; r := r'; l := l')\\ \quad\ \mathsf{else}\ \ (r := r'; \mathsf{skip}; l := l')\end{array}$$

when analyzing security w.r.t. an observer with domain *left*. Lifting for *right* then results in

$$\begin{array}{l}\mathsf{if}\ t\ \mathsf{then}\ \ (t := t'; \alpha_1; r := r'; l := l'; \alpha_2)\\ \quad\ \mathsf{else}\ \ (\alpha_3; r := r'; \mathsf{skip}; l := l'; \alpha_4).\end{array}$$

Unification and projection gives

$$\begin{array}{l}\mathsf{if}\ t\ \mathsf{then}\ \ (t := t'; r := r'; l := l'; \mathsf{skip})\\ \quad\ \mathsf{else}\ \ (\mathsf{skip}; r := r'; \mathsf{skip}; l := l')\end{array}$$

Observe that this program is not secure any more from the viewpoint of a *left*-observer. Applying the transformation again for domain *left* results in the secure program

$$\begin{array}{l}\mathsf{if}\ t\ \mathsf{then}\ (t := t'; r := r'; \mathsf{skip}; l := l'; \mathsf{skip})\\ \quad\ \mathsf{else}\ \ (\mathsf{skip}; r := r'; \mathsf{skip}; l := l'; \mathsf{skip}),\end{array}$$

which is a fixpoint of both transformations.

Note that the idempotence of the transformation is necessary (but not sufficient) for the existence of a fixpoint and, hence, for the termination of such an iterative approach. As is illustrated in Example 12, the transformation realized by our type system is idempotent, whereas the transformation from [14] is not.

Another possibility to tackle multi-level security policies in our setting is to unify the branches of a conditional with guard of security level D' under the theory $\bigcap_{D \not\geq D'} \doteq_D$. This would result in a multi-level, transforming security type system that supports a single-pass transformation. However, an investigation of this possibility remains to be done.

7 Conclusions

We proposed a novel approach to analyzing the security of information flow in concrete programs with the help of transforming security type systems where the key idea has been to integrate unification with typing rules. This yielded a very natural perspective on the problem of eliminating inter-command information leakage.

We instantiated our approach by defining a program equivalence that captures the behavioral equivalence to be preserved during the transformation and an observational equivalence that captures the perspective of a low-level attacker. This led to a novel transforming security type system and calculi for automatically inserting meta-variables into programs and for computing substitutions. We proved that the resulting analysis technique is sound and also provided a completeness result. The main advantages of our approach include that the precision of type checking is improved, that additional insecure programs can be corrected, that the resulting programs are faster and smaller in size, and that it offers the possibility to analyze programs under security policies with more than two security domains.

It will be interesting to see how our approach performs for other choices of the parameters like, e.g., observational equivalences that admit intentional declassification [9]. Another interesting possibility is to perform the entire information flow analysis and program transformation using unification without any typing rules, which would mean to further explore the possibilities of the PER model. Finally, it would be desirable to integrate our fully automatic transformation into an interactive framework for supporting the programmer in correcting insecure programs as sketched in the introduction.

Acknowledgments We thank the anonymous reviewers for valuable comments. We also thank Daniel Hedin and Paul Hankes Drielsma for their feedback to this and earlier versions of the article.

A Proofs of the Technical Results

A.1 Proof of Theorem 1

Theorem 1 *If $V \doteq_L V'$ is derivable then $V \approx_L V'$ holds.*

Before proving Theorem 1, we introduce a lemma and prove it using the bisimulation-up-to technique.

Definition 9 A binary relation R on commands is a *strong low bisimulation up to* $\approx_L$ if R is symmetric and

$$\begin{array}{l}\forall C, C', C_1, \ldots, C_n \in Com.\ \forall s, s', t \in S:\\ \quad (C\ R\ C' \wedge s =_L s' \wedge \langle\!| C, s |\!\rangle \twoheadrightarrow \langle\!| C_1 \ldots C_n, t |\!\rangle)\\ \quad \Rightarrow \exists C'_1, \ldots, C'_n \in Com.\ \exists t' \in S.\ (\langle\!| C', s' |\!\rangle \twoheadrightarrow \langle\!| C'_1 \ldots C'_n, t' |\!\rangle\\ \qquad \wedge\ \forall i \in \{1, \ldots, n\} : C_i (R \cup \approx_L)^+ C'_i \wedge t =_L t')\end{array}$$

Theorem 7 *If R is a strong low bisimulation up to $\approx_L$ then $R \subseteq \approx_L$ holds.*

[8] Transforming a program in multiple passes would not be necessary if a multi-level version of the given type system were available. However, multi-level versions do not exist so far for contemporary transforming security type systems.

Proof (Sketch) Define $Q = ((R \cup \approx_L)^{\uparrow})^{+}$ where $^{+}$ returns the transitive closure of a relation and $^{\uparrow}$ returns the pointwise lifting of a relation on commands to command vectors (here, $R \cup \approx_L$ is viewed as a relation on commands). We obtain $Q \subseteq \approx_L$ from the fact that Q satisfies the implication in Definition 1 (replacing R in the definition by Q).

The implication can be proved by a straightforward induction over the Q-distance between two Q-related programs V and V', where the *Q-distance* between V and V' is the minimal length of a sequence $V_1, \ldots, V_n$ with the properties $\forall i \in \{0, \ldots, n\}\colon V_i\ (R \cup \approx_L)^{\uparrow}\ V_{i+1}$, $V_0 = V$, and $V_{n+1} = V'$. Such a sequence exists because $V\ Q\ V'$ holds. We obtain $R \subseteq \approx_L$ from $R \subseteq Q$ (by definition of Q) and the transitivity of $\subseteq$. □

Lemma 5

1. *If* $Id : high$ *then* skip $\approx_L Id := Exp$.
2. *If* $Exp, Exp' : low$ *and* $Exp \equiv Exp'$ *then* $Id := Exp \approx_L Id := Exp'$.
3. *If* $C_1 \approx_L C_1'$ *and* $C_2 \approx_L C_2'$ *then* $C_1; C_2 \approx_L C_1'; C_2'$.
4. *If* $C_1 \approx_L C_1'$ *and* $V_2 \approx_L V_2'$ *then* fork$(C_1 V_2) \approx_L$ fork$(C_1' V_2')$.
5. *If* $B, B' : low$, $B \equiv B'$, *and* $C_1 \approx_L C_1'$ *then* while B do $C_1 \approx_L$ while B' do C_1'.
6. *If* $B, B' : low$, $B \equiv B'$, $C_1 \approx_L C_1'$, *and* $C_2 \approx_L C_2'$ *then* if B then C_1 else $C_2 \approx_L$ if B' then C_1' else C_2'.
7. *If* $C_1 \approx_L C_1'$ *and* $C_1 \approx_L C_2'$ *then* skip; $C_1 \approx_L$ if B' then C_1' else C_2'.

Proof We illustrate the proof idea with three example cases. In each case, we prove the strong low bisimilarity of the two commands with the bisimulation up-to technique. That is, we define a binary relation R on commands that relates the two commands and prove that R is a strong low bisimulation up to $\approx_L$. From Theorem 7, we then obtain that the two given commands are strongly low bisimilar.

1. Define R as the symmetric closure of the relation $\{(\mathsf{skip}, Id := Exp) \mid Id : high\}$. Let $(\mathsf{skip}, Id := Exp) \in R$ and $s, s' \in S$ be arbitrary with $s =_L s'$. From the operational semantics, we obtain $\langle\!|\mathsf{skip}, s|\!\rangle \rightarrowtriangle \langle\!|\langle\rangle, s|\!\rangle$. Moreover, there is a $t' \in S$ such that $\langle\!|Id := Exp, s'|\!\rangle \rightarrowtriangle \langle\!|\langle\rangle, t'|\!\rangle$. From $s =_L s'$ and $Id : high$, we obtain $s =_L t'$. Let $(Id := Exp, \mathsf{skip}) \in R$ and $s, s', t \in S$ be arbitrary with $s =_L s'$ and $\langle\!|Id := Exp, s|\!\rangle \rightarrowtriangle \langle\!|\langle\rangle, t|\!\rangle$. We have $\langle\!|\mathsf{skip}, s'|\!\rangle \rightarrowtriangle \langle\!|\langle\rangle, s'|\!\rangle$. From $s =_L s'$ and $Id : high$, we obtain $t =_L s'$.
   Hence, R is a strong low bisimulation up to $\approx_L$.
2. Define R as the symmetric relation $\{(C_1; C_2, C_1'; C_2') \mid C_1 \approx_L C_1', C_2 \approx_L C_2'\}$.
   Let $(C_1; C_2, C_1'; C_2') \in R$ and $s, s', t \in S$ be arbitrary with $s =_L s'$ and $\langle\!|C_1; C_2, s|\!\rangle \rightarrowtriangle \langle\!|C^*, t|\!\rangle$ for some $C^* \in \boldsymbol{Com}$. We make a case distinction on C^* according to the operational semantics:
   (a) $C^* = C_2$: From the operational semantics, we obtain $\langle\!|C_1, s|\!\rangle \rightarrowtriangle \langle\!|\langle\rangle, t|\!\rangle$. Since $C_1 \approx_L C_1'$ and $s =_L s'$, there is a $t' \in S$ with $\langle\!|C_1', s'|\!\rangle \rightarrowtriangle \langle\!|\langle\rangle, t'|\!\rangle$ and $t =_L t'$ according to Definition 1. From the operational semantics, we obtain $\langle\!|C_1'; C_2', s'|\!\rangle \rightarrowtriangle \langle\!|C_2', t'|\!\rangle$ with $C_2 \approx_L C_2'$ (by definition of R) and $t =_L t'$.
   (b) $C^* = \langle C; C_2\rangle V$ for some $C \in Com$ and $V \in \boldsymbol{Com}$ (possibly $V = \langle\rangle$). From the operational semantics, we obtain $\langle\!|C_1, s|\!\rangle \rightarrowtriangle \langle\!|\langle C\rangle V, t|\!\rangle$. Since $C_1 \approx_L C_1'$ and $s =_L s'$, there are $C' \in Com$, $V' \in \boldsymbol{Com}$, and $t' \in S$ with $\langle\!|C_1', s'|\!\rangle \rightarrowtriangle \langle\!|\langle C'\rangle V', t'|\!\rangle$, $C \approx_L C'$, $V \approx_L V'$, and $t =_L t'$. From the operational semantics, we obtain $\langle\!|C_1'; C_2', s'|\!\rangle \rightarrowtriangle \langle\!|\langle C'; C_2'\rangle V', t'|\!\rangle$ with $(C; C_2, C'; C_2') \in R$ (follows from $C \approx_L C'$, $C_2 \approx_L C_2'$, and the definition of R), $V \approx_L V'$, and $t =_L t'$.
   Hence, R is a strong low bisimulation up to $\approx_L$.
3. Define R as the symmetric relation

$$\{\ (\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \mid B, B' : low, B \equiv B', C_1 \approx_L C_1', C_2 \approx_L C_2'\}$$

   Let $(\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \in R$ and $s, s' \in S$ be arbitrary with $s =_L s'$. We make a case distinction on the value of B in s:
   (a) $\langle\!|B, s|\!\rangle \downarrow \mathsf{False}$: From $s =_L s'$, $B, B' : low$, and $B \equiv B'$, we obtain $\langle\!|B', s'|\!\rangle \downarrow \mathsf{False}$. From the operational semantics, we obtain $\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s|\!\rangle \rightarrowtriangle \langle\!|C_2, s|\!\rangle$ and $\langle\!|\mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2', s'|\!\rangle \rightarrowtriangle \langle\!|C_2', s'|\!\rangle$ with $s =_L s'$. By definition of R, we have $C_2 \approx_L C_2'$.
   (b) $\langle\!|B, s|\!\rangle \downarrow \mathsf{True}$: From $s =_L s'$, $B, B' : low$, and $B \equiv B'$, we obtain $\langle\!|B', s'|\!\rangle \downarrow \mathsf{True}$. From the operational semantics, we obtain $\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s|\!\rangle \rightarrowtriangle \langle\!|C_1, s|\!\rangle$ and $\langle\!|\mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2', s'|\!\rangle \rightarrowtriangle \langle\!|C_1', s'|\!\rangle$ with $s =_L s'$. By definition of R, we have $C_1 \approx_L C_1'$.
   Hence, R is a strong low bisimulation up to $\approx_L$. □

We are now ready to prove Theorem 1.

Proof of Theorem 1 The proof proceeds by induction on the number of rule applications in the derivation $\mathcal{D}$ of $V \mathrel{\hat{=}}_L V'$.

Base case: $\mathcal{D}$ consists of only a single rule application. We make a case distinction on this rule.

[*Skip*] The judgment derived is skip $\mathrel{\hat{=}}_L$ skip. Lemma 5(1) implies skip $\approx_L h := h$ where h is an

arbitrary variable with $h : high$. From symmetry and transitivity of $\approx_L$, we obtain skip $\approx_L$ skip.

[SHA_1] The judgment derived is skip $\doteq_L Id := Exp$ with $Id : high$. From Lemma 5(1) follows skip $\approx_L Id := Exp$.

[SHA_2] The judgment derived is $Id := Exp \doteq_L$ skip with $Id : high$. From Lemma 5(1) and the symmetry of $\approx_L$, we obtain $Id := Exp \approx_L$ skip.

[HA] The judgment derived is $Id := Exp \doteq_L Id' := Exp'$ with $Id, Id' : high$. From Lemma 5(1), we obtain skip $\approx_L Id := Exp$ and skip $\approx_L Id' := Exp'$. Symmetry and transitivity of $\approx_L$ imply $Id := Exp \approx_L Id' := Exp'$.

[LA] The judgment derived is $Id := Exp \doteq_L Id := Exp'$ with $Id : low$, $Exp, Exp' : low$, and $Exp \equiv Exp'$. Lemma 5(2) implies $Id := Exp \approx_L Id := Exp'$.

Induction assumption: If $\mathcal{D}'$ is a derivation of $W \doteq_L W'$ with fewer rule applications than in $\mathcal{D}$ then $W \approx_L W'$ holds.

Step case: We make a case distinction depending on the rule applied at the root of $\mathcal{D}$.

[$SComp$] The judgment derived is $C_1; C_2 \doteq_L C_1'; C_2'$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_2'$, respectively. From the induction assumption, we obtain $C_1 \approx_L C_1'$ and $C_2 \approx_L C_2'$. Lemma 5(3) implies $C_1; C_2 \approx_L C_1'; C_2'$.

[$PComp$] The judgment derived is

$$\langle C_1, \ldots, C_n \rangle \doteq_L \langle C_1', \ldots, C_n' \rangle$$

and there are derivations $\mathcal{D}_i$ of $C_i \doteq_L C_i'$ for $i = 1, \ldots, n$. From the induction assumption, we obtain $C_i \approx_L C_i'$ for $i = 1, \ldots, n$. From Definition 1, we obtain $\langle C_1, \ldots, C_n \rangle \approx_L \langle C_1', \ldots, C_n' \rangle$.

[$Fork$] The judgment derived is fork$(C_1 V_1) \doteq_L$ fork$(C_1' V_1')$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $V_1 \doteq_L V_1'$, respectively. From the induction assumption, we obtain $C_1 \approx_L C_1'$ and $V_1 \approx_L V_1'$. Lemma 5(4) implies fork$(C_1 V_1) \approx_L$ fork$(C_1' V_1')$.

[WL] The judgment derived is

$$\text{while } B \text{ do } C_1 \doteq_L \text{while } B' \text{ do } C_1'$$

with $B, B' : low$, $B \equiv B'$, and there is a derivation $\mathcal{D}_1$ of $C_1 \doteq_L C_1'$. Lemma 5(5) implies then that while B do $C_1 \approx_L$ while B' do C_1'.

[$LIte$] The judgment derived is

$$\text{if } B \text{ then } C_1 \text{ else } C_2 \doteq_L \text{if } B' \text{ then } C_1' \text{ else } C_2'$$

with $B, B' : low$, $B \equiv B'$, and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_2'$, respectively. Lemma 5(6) implies if B then C_1 else $C_2 \approx_L$ if B' then C_1' else C_2'.

[$SHIte_1$] The judgment derived is

$$\text{skip}; C_1 \doteq_L \text{if } B' \text{ then } C_1' \text{ else } C_2'$$

with $B' : high$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_1 \doteq_L C_2'$, respectively. From the induction assumption, we obtain $C_1 \approx_L C_1'$ and $C_1 \approx_L C_2'$. Lemma 5(7) implies skip; $C_1 \approx_L$ if B' then C_1' *else* C_2'.

[$SHIte_2$] The judgment derived is

$$\text{if } B \text{ then } C_1 \text{ else } C_2 \doteq_L \text{skip}; C_1'$$

with $B : high$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_1'$, respectively. From the induction assumption, we obtain $C_1 \approx_L C_1'$ and $C_2 \approx_L C_1'$. Symmetry of $\approx_L$ and Lemma 5(7) imply skip; $C_1' \approx_L$ if B then C_1 else C_2. From the symmetry of $\approx_L$, we obtain if B then C_1 else $C_2 \approx_L$ skip; C_1'.

[$HAHIte_1$] The judgment derived is

$$Id := Exp; C_1 \doteq_L \text{if } B' \text{ then } C_1' \text{ else } C_2'$$

with $Id : high$, $B' : high$, and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_1 \doteq_L C_2'$, respectively. From skip; $C_1 \approx_L$ if B' then C_1' else C_2' (see Case [$SHIte_1$]), $Id := Exp; C_1 \approx_L$ skip; C_1 (see Case [SHA_2]), and transitivity of $\approx_L$, we obtain $Id := Exp; C_1 \approx_L$ if B' then C_1' else C_2'.

[$HAHIte_2$] The judgment derived is if B then C_1 else $C_2 \doteq_L Id' := Exp'; C_1'$ with $Id' : high$, $B : high$, and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_1'$, respectively. From if B then C_1 else $C_2 \approx_L$ skip; C_1' (see Case [$SHIte_2$]), skip; $C_1' \approx_L Id' := Exp'; C_1'$ (see Case [SHA_1]), and transitivity of $\approx_L$, we obtain if B then C_1 else $C_2 \approx_L Id' := Exp'; C_1'$.

[$HIte$] The judgment derived is if B then C_1 else $C_2 \doteq_L$ if B' then C_1' else C_2' with $B, B' : high$ and there are derivations $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$ of $C_1 \doteq_L C_1'$, $C_1 \doteq_L C_2'$, and $C_1 \doteq_L C_2$, respectively. From the induction assumption, we obtain $C_1 \approx_L C_1'$, $C_1 \approx_L C_2'$, and $C_1 \approx_L C_2$. Symmetry and transitivity of $\approx_L$ implies $C_1 \approx_L C_1$. From if B then C_1 else $C_2 \approx_L$ skip; C_1 (see Case [$SHIte_2$]), skip; $C_1 \approx_L$ if B' then C_1' else C_2' (see Case [$SHIte_1$]), and transitivity of $\approx_L$, we then obtain if B then C_1 else $C_2 \approx_L$ if B' then C_1' else C_2'. □

A.2 Proof of Theorem 2

Theorem 2

1. *For all preserving substitutions σ, ρ that are ground for $V \in \boldsymbol{Com_{\mathcal{V}}}$, we have $\sigma(V) \simeq \rho(V)$.*
2. *For each lifting V' of a ground program $V \in \boldsymbol{Com}$ and each preserving substitution σ with $\sigma(V')$ ground, we have $\sigma(V') \simeq V$.*

For the proof of Theorem 2 we first strengthen our notion of bisimulation. Then we prove a lemma that

shows that this relation is a congruence by using the up-to technique.

Definition 10 The *pointwise weak possibilistic bisimulation* $\dot{\simeq}$ is the union of all symmetric relations R on command vectors $V, V' \in \boldsymbol{Com}$ of equal size, i.e., $V = \langle C_1, \ldots, C_n\rangle, V' = \langle C'_1, \ldots, C'_n\rangle$, such that whenever VRV' then for all states s, t and all $i \in \{1, \ldots, n\}$ and all thread pools W there is a thread pool W' with

$$\begin{array}{l}\langle\!| C_i, s|\!\rangle \rightarrowtriangle \langle\!| W, t|\!\rangle \Rightarrow (\langle\!| C'_i, s|\!\rangle \rightarrowtriangle^* \langle\!| W', t|\!\rangle \wedge W\,R\,W') \\ \text{and } V = \langle\rangle \implies \langle\!| V', s|\!\rangle \rightarrow^* \langle\!|\langle\rangle, s|\!\rangle.\end{array} \quad (1)$$

Observe that Property 1 also holds for the entire relation $\dot{\simeq}$. Furthermore, for two thread pools $V = \langle C_1, \ldots, C_n\rangle$, $V' = \langle C'_1, \ldots, C'_n\rangle$ we have $V \dot{\simeq} V'$ if and only if for all $i \in \{1, \ldots, n\}$ we have $C_i \dot{\simeq} C'_i$.

Lemma 6 $V \dot{\simeq} V' \Rightarrow V \simeq V'$

Proof Follows directly from Definitions 2 and 10 and the operational semantics for thread pools. □

Definition 11 A binary relation R on commands is a *pointwise weak possibilistic bisimulation up to* $\dot{\simeq}$ if R is symmetric and

$$\begin{array}{l}\forall C, C', C_1, \ldots, C_n \in \mathit{Com}.\ \forall s, t \in S.\\ \quad (C\ R\ C' \wedge \langle\!| C, s|\!\rangle \rightarrowtriangle \langle\!| C_1 \ldots C_n, t|\!\rangle)\\ \quad \Rightarrow \exists C'_1, \ldots, C'_n \in \mathit{Com}.\ (\langle\!| C', s|\!\rangle \rightarrowtriangle^* \langle\!| C'_1 \ldots C'_n, t|\!\rangle\\ \qquad \wedge\ \forall i \in \{1, \ldots, n\} : C_i (R \cup \dot{\simeq}) C'_i).\end{array}$$

Theorem 8 *If R is a pointwise weak possibilistic bisimulation up to $\dot{\simeq}$, then we have $R \subseteq \dot{\simeq}$.*

Proof Let $Q = (R \cup \dot{\simeq})^{\uparrow}$, where $^{\uparrow}$ returns the pointwise lifting of a relation on commands to command vectors (here, $R \cup \dot{\simeq}$ is viewed as a relation on commands). It is sufficient to show $Q \subseteq \dot{\simeq}$. To this end, we show that Q satisfies the condition in Definition 10, and is therefore contained in $\dot{\simeq}$, the union of all such relations. Let $V = \langle C_1, \ldots, C_n\rangle$ and $V' = \langle C'_1, \ldots, C'_n\rangle$ and $(V, V') \in Q$. If $n = 0$, then we have $V = V' = \langle\rangle$ and the second condition of Definition 10 is fulfilled. Suppose $n > 0$ and $\langle\!| V, s|\!\rangle \rightarrowtriangle \langle\!| W, t|\!\rangle$. By definition of the operational semantics, we know that there is $i \in \{1, \ldots, n\}$ with $\langle\!| C_i, s|\!\rangle \rightarrowtriangle \langle\!| C_{i,1}, \ldots, C_{i,m}, t|\!\rangle$ and $W = \langle C_1, \ldots, C_{i-1}, C_{i,1}, \ldots, C_{i,m}, C_{i+1}, \ldots, C_n\rangle$.

Case 1: $(C_i, C'_i) \in \dot{\simeq}$. According to Definition 10, there are $C'_{i,1}, \ldots, C'_{i,m}$ with $\langle\!| C'_i, s|\!\rangle \rightarrowtriangle^* \langle\!| C'_{i,1}, \ldots, C'_{i,m}, t|\!\rangle$ and $\langle C_{i,1}, \ldots, C_{i,m}\rangle \dot{\simeq} \langle C'_{i,1}, \ldots, C'_{i,m}\rangle$.

From Definition 10 it follows that $C_{i,j} \dot{\simeq} C'_{i,j}$ holds for all $j \in \{1, \ldots, m\}$.

Case 2: $(C_i, C'_i) \in R$. By Definition 11, there are $C'_{i,1}, \ldots, C'_{i,m}$ with $\langle\!| C'_i, s|\!\rangle \rightarrowtriangle^* \langle\!| C'_{i,1}, \ldots, C'_{i,m}, t|\!\rangle$ and $C_{i,j} (R \cup \dot{\simeq}) C'_{i,j}$ for all $j \in \{1, \ldots, m\}$.

With $W' = \langle C'_1, \ldots, C'_{i-1}, C'_{i,1}, \ldots, C'_{i,m}, C'_{i+1}, \ldots, C'_n\rangle$ we have $\langle\!| V', s|\!\rangle \rightarrowtriangle^* \langle\!| W', t|\!\rangle$ and $(W, W') \in Q$. As $\dot{\simeq}$ is defined to be the union of all symmetric relations with the property of Definition 10, we see $Q \subseteq \dot{\simeq}$. □

Lemma 7

1. *If $C_1 \dot{\simeq} C'_1$ and $C_2 \dot{\simeq} C'_2$ then $C_1; C_2 \dot{\simeq} C'_1; C'_2$.*
2. *If $C_1 \dot{\simeq} C'_1$ and $V_2 \dot{\simeq} V'_2$ then $\mathsf{fork}(C_1 V_2) \dot{\simeq} \mathsf{fork}(C'_1 V'_2)$.*
3. *If $C_1 \dot{\simeq} C'_1$ then $\mathsf{while}\ B\ \mathsf{do}\ C_1 \dot{\simeq} \mathsf{while}\ B\ \mathsf{do}\ C'_1$.*
4. *If $C_1 \dot{\simeq} C'_1$ and $C_2 \dot{\simeq} C'_2$ then $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \dot{\simeq} \mathsf{if}\ B\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2$.*
5. *If $C_1 \dot{\simeq} C'_1, \ldots, C_n \dot{\simeq} C'_n$ then $\langle C_1, \ldots, C_n\rangle \dot{\simeq} \langle C'_1, \ldots, C'_n\rangle$.*

Proof We proceed as in the proof of Lemma 5, only by using pointwise weak possibilistic bisimulations up to $\dot{\simeq}$ instead of strong low-bisimulations. □

Proof of Theorem 2 We prove the first assertion for pointwise weak possibilistic bisimulations (rather than weak possibilistic bisimulations) by induction on the term structure of vectors of length 1, i.e., $V \in \mathit{Com}_{\mathcal{V}}$. By Lemma 7(5), the assertion is then lifted to arbitrary vectors in $\boldsymbol{Com}_{\boldsymbol{\mathcal{V}}}$, and with Lemma 6, we obtain what we wanted. Let σ, ρ be substitutions that are preserving and ground for V.

1. Suppose V is skip or an assignment. Then $\sigma(V) = \rho(V) = V$, and the assertion follows by reflexivity of $\dot{\simeq}$.
2. Suppose V is of the form $\alpha; C'$ or $C'; \alpha$. By induction hypothesis we have $\sigma(C') \dot{\simeq} \rho(C')$. σ and ρ are preserving and ground for α, so we see $\sigma(\alpha) \dot{\simeq} \rho(\alpha)$. From Lemma 7(1), $\sigma V \dot{\simeq} \rho V$ follows.
3. Suppose $V = C_1; C_2$ with $C_1, C_2 \in \mathit{Com}_{\mathcal{V}}$. By induction hypothesis we have $\sigma C_1 \dot{\simeq} \rho C_1$ and $\sigma C_2 \dot{\simeq} \rho C_2$. From Lemma 7(1), $\sigma V \dot{\simeq} \rho V$ follows.
4. Suppose $V = \mathsf{while}\ B\ \mathsf{do}\ C'$ with $C' \in \mathit{Com}_{\mathcal{V}}$. By induction hypothesis we have $\sigma C' \dot{\simeq} \rho C'$. From Lemma 7(3), $\sigma V \dot{\simeq} \rho V$ follows.
5. Suppose $V = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ with $C_1, C_2 \in \mathit{Com}_{\mathcal{V}}$. By induction hypothesis we have $\sigma C_1 \dot{\simeq} \rho C_1$ and $\sigma C_2 \dot{\simeq} \rho C_2$. From Lemma 7(4), $\sigma V \dot{\simeq} \rho V$ follows.

6. Suppose $V = \mathsf{fork}(C_0\langle C_1,\ldots,C_n\rangle)$ with $C_i \in Com_{\mathcal{V}}$ for $i = 0,\ldots,n$. By induction hypothesis we have $\sigma C_i \doteq \rho C_i$ for $i = 0,\ldots,n$. From Lemma 7(5), we first obtain $\sigma\langle C_1,\ldots,C_n\rangle \doteq \rho\langle C_1,\ldots,C_n\rangle$ and then, by Lemma 7(2), $\sigma V \doteq \rho V$.

The second assertion follows directly from part 1.

A.3 Proof of Theorem 3

Theorem 3 *If $V \hookrightarrow V' : S$ can be derived then*

1. *V' has secure information flow,*
2. *$V \simeq V'$ holds, and*
3. *$V' \approxeq_L S$ holds.*

We first state and prove two lemmas to simplify reasoning with $\doteq_L$ on $Com_{\mathcal{V}}$.

Lemma 8 *If $V_1 \doteq_L V_2$ holds for two programs $V_1, V_2 \in \boldsymbol{Com_{\mathcal{V}}}$ then $\sigma V_1 \doteq_L \sigma V_2$ holds for each substitution σ that is preserving (but not necessarily ground).*

Proof Given an arbitrary substitution η that is preserving and ground for σV_1 and σV_2, we obtain $\eta(\sigma V_1) \doteq_L \eta(\sigma V_2)$ from $V_1 \doteq_L V_2$, Definition 5, and the fact that $\eta \circ \sigma$ is preserving and ground for V_1 and V_2. Since η was chosen arbitrarily, $\sigma V_1 \doteq_L \sigma V_2$ follows. □

Lemma 9 *Let $V, V', V_0, V_0', \ldots, V_n, V_n' \in \boldsymbol{Com_{\mathcal{V}}}$ be programs that may contain meta-variables. If $V_i \doteq_L V_i'$ holds for each $i \in \{0,\ldots,n\}$ according to Definition 5 and $V \doteq_L V'$ can be syntactically derived from the assumptions $V_0 \doteq_L V_0'$, …, $V_n \doteq_L V_n'$ with the rules in Figure 3 then $V \doteq_L V'$ holds according to Definition 5.*

Proof We argue by induction on the size of $\mathcal{D}$, the derivation of $V \doteq_L V'$ from $V_0 \doteq_L V_0'$, …, $V_n \doteq_L V_n'$.

Base case: If $\mathcal{D}$ consists of zero rule applications then $V \doteq_L V'$ equals one of the assumptions.

Induction assumption: The proposition holds for every derivation with less than n rule applications.

Step case: Assume $\mathcal{D}$ consists of n rule applications. We make a case distinction on the last rule applied in $\mathcal{D}$. Here, we consider only the case where [*SComp*] is the last rule applied. The reasoning principle is independent of the structure of the rule [*SComp*], and so the cases for the other rules can be shown along the same lines.

Let $C_1, C_1', C_2, C_2' \in Com_{\mathcal{V}}$ be arbitrary with $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_2'$. Let σ be an arbitrary substitution that is preserving and ground for C_1, C_1', C_2, C_2'. From $C_1 \doteq_L C_1'$, $C_2 \doteq_L C_2'$, and Definition 5 we obtain $\sigma C_1 \doteq_L \sigma C_1'$ and $\sigma C_2 \doteq_L \sigma C_2'$. An application of [*SComp*] (for ground programs) yields $(\sigma C_1; \sigma C_2) \doteq_L (\sigma C_1'; \sigma C_2')$. Since $\sigma(C_1; C_2) = (\sigma C_1; \sigma C_2)$, $(\sigma C_1'; \sigma C_2') = \sigma(C_1'; C_2')$, and σ was chosen freely, we obtain $C_1; C_2 \doteq_L C_1'; C_2'$ from Definition 5. □

Proof of Theorem 3 We prove the three propositions in different order.

2. By induction on the height of the given derivation of $V \hookrightarrow V' : S$, one obtains $V' = \rho V$ for some preserving substitution ρ. The assertion follows by applying Theorem 2.1 to $\sigma(V)$ and $(\sigma\rho)(V)$ for an arbitrary σ that is preserving and ground for both V and V'.
3. Since $V' \doteq_L S$ implies $V' \approxeq_L S$ according to Theorem 1, it suffices to show that $V \hookrightarrow V' : S$ implies $V' \doteq_L S$. We prove this second proposition by induction on the minimal height of the given derivation $\mathcal{D}$ of $V \hookrightarrow V' : S$.
   Base case: $\mathcal{D}$ consists of a single rule application. We perform a case distinction on this rule:
   [*Var*] We have $V = V' = S = \alpha$ for some meta-variable $\alpha \in \mathcal{V}$. Let σ be an arbitrary substitution that is preserving and ground for α. As $\sigma\alpha$ is a command in $Stut_{\mathcal{V}}$ that is free of meta-variables (i.e., a sequential composition of skip statements), we obtain $\sigma\alpha \doteq_L \sigma\alpha$ from [*Skip*] and [*SComp*] in Fig. 3. Hence, $\alpha \doteq_L \alpha$ holds.
   [*Skp*] We have $V = V' = S = \mathsf{skip}$. From [*Skip*] in Fig. 3, we obtain $\mathsf{skip} \doteq_L \mathsf{skip}$.
   [Ass_h] We have $V = V' = Id := Exp$ and $S = \mathsf{skip}$ with $Id : high$. From [SHA_2] in Fig. 3, we obtain $Id := Exp \doteq_L \mathsf{skip}$.
   [Ass_l] We have $V = V' = S = Id := Exp$ with $Id : low$ and $Exp : low$. From [*LA*] in Fig. 3, we obtain $Id := Exp \doteq_L Id := Exp$.
   Induction assumption: For any derivation $\mathcal{D}'$ of a judgment $W \hookrightarrow W' : S'$ with height less than the height of $\mathcal{D}$, $W' \doteq_L S'$ holds.
   Step case: We make a case distinction on the rule applied at the root of $\mathcal{D}$.
   [*Seq*] We have $V = C_1; C_2$, $V' = C_1'; C_2'$, and $S = S_1; S_2$ with $C_1 \hookrightarrow C_1' : S_1$ and $C_2 \hookrightarrow C_2' : S_2$. By induction assumption, $C_1' \doteq_L S_1$ and $C_2' \doteq_L S_2$ hold. An application of [*SComp*] in Fig. 3 yields $C_1'; C_2' \doteq_L S_1; S_2$.
   [*Par*] We have $V = \langle C_1,\ldots,C_n\rangle$, $V' = \langle C_1',\ldots,C_n'\rangle$, and $S = \langle S_1,\ldots,S_n\rangle$ with $C_i \hookrightarrow C_i' : S_i$ for all $i \in \{1,\ldots,n\}$. By induction assumption, $C_i' \doteq_L S_i$ holds for all $i \in \{1,\ldots,n\}$. Application of [*PComp*] in Fig. 3 yields $\langle C_1',\ldots,C_n'\rangle \doteq_L \langle S_1,\ldots,S_n\rangle$.
   [*Frk*] We have $V = \mathsf{fork}(C_1 V_2)$, $V' = \mathsf{fork}(C_1' V_2')$, and $S = \mathsf{fork}(S_1 S_2)$ with $C_1 \hookrightarrow C_1' : S_1$ and $V_2 \hookrightarrow V_2' : S_2$. By induction assumption, $C_1' \doteq_L S_1$ and

$V_2' \doteq_L S_2$ hold. An application of [*Fork*] in Fig. 3 yields $\mathsf{fork}(C_1'V_2') \doteq_L \mathsf{fork}(S_1S_2)$.
[*Whl*] We have $V = \mathsf{while}\ B\ \mathsf{do}\ C_1$, $V' = \mathsf{while}\ B\ \mathsf{do}\ C_1'$, and $S = \mathsf{while}\ B\ \mathsf{do}\ S_1$ with $B : \mathit{low}$ and $C_1 \hookrightarrow C_1' : S_1$. By induction assumption, $C_1' \doteq_L S_1$ holds. An application of [*WL*] in Fig. 3 yields $\mathsf{while}\ B\ \mathsf{do}\ C_1' \doteq_L \mathsf{while}\ B\ \mathsf{do}\ S_1$.
[Cond_l] We have $V = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, together with $V' = \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ and $S = \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ with $B : \mathit{low}$, $C_1 \hookrightarrow C_1' : S_1$, and $C_2 \hookrightarrow C_2' : S_2$. By induction assumption, $C_1' \doteq_L S_1$ and $C_2' \doteq_L S_2$ hold. An application of [*LIte*] in Fig. 3 yields $\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$.
[Cond_h] We have $V = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, together with $V' = \mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2'$ and $S = \mathsf{skip}; \sigma S_1$ with $B : \mathit{high}$, $C_1 \hookrightarrow C_1' : S_1$, $C_2 \hookrightarrow C_2' : S_2$, and $\sigma \in \mathcal{U}(\{S_1 \doteq_L^? S_2\})$. By induction assumption, $C_1' \doteq_L S_1$ and $C_2' \doteq_L S_2$ hold. From Lemma 8, we obtain $\sigma C_1' \doteq_L \sigma S_1$ and $\sigma C_2' \doteq_L \sigma S_2$ as σ is preserving. Then we obtain $\sigma C_2' \doteq_L \sigma S_1$ from $\sigma C_2' \doteq_L \sigma S_2$, $\sigma S_1 \doteq_L \sigma S_2$ (follows from $\sigma \in \mathcal{U}(\{S_1 \doteq_L^? S_2\})$), and the fact that $\doteq_L$ is symmetric and transitive. An application of [SHIte_2] in Fig. 3 yields $\mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2' \doteq_L \mathsf{skip}; \sigma S_1$.

1. From $V' \approx_L S$ and the symmetry of $\approx_L$, we obtain $S \approx_L V'$. Then $V' \approx_L V'$ follows from the transitivity of $\approx_L$. □

A.4 Proof of Lemma 1

Lemma 1 *For two commands C_1 and C_2 in $\mathit{Pad}_\mathcal{V}$ we have $C_1 \doteq_L C_2$ if and only if $\mathit{const}(C_1) = \mathit{const}(C_2)$ and $\forall \alpha \in \mathcal{V}: |C_1|_\alpha = |C_2|_\alpha$.*

We define $\mathit{const}(C) = |C|_{\mathsf{skip}} + \sum_{Id_h, Exp, Id_h : high} |C|_{Id_h := Exp}$, where $|C|_D$ denotes the number of occurrences of D as a subterm of C.

Proof of Lemma 1 We show each direction of the implication:
($\Longrightarrow$) Assume $C_1 \doteq_L C_2$. Make a case distinction:

1. Assume $\mathit{const}(C_1) \neq \mathit{const}(C_2)$. Let σ be the substitution mapping all variables in C_1 and C_2 to ϵ. The judgment $\sigma C_1 \doteq_L \sigma C_2$ is not derivable with the rules in Fig. 3. According to Definition 5, this contradicts the assumption $C_1 \doteq_L C_2$, as σ is preserving. Hence, this case is not possible.
2. Assume $\mathit{const}(C_1) = \mathit{const}(C_2)$ and $|C_1|_\alpha \neq |C_2|_\alpha$ for some meta-variable $\alpha \in \mathcal{V}$. Let σ be the substitution mapping α to skip and all other variables in C_1 and C_2 to ϵ. We have $\mathit{const}(\sigma C_1) \neq \mathit{const}(\sigma C_2)$ and, thus, a contradiction to the assumption $C_1 \doteq_L C_2$ (argue like in case 1).
3. Assume $\mathit{const}(C_1) = \mathit{const}(C_2)$ and $|C_1|_\alpha = |C_2|_\alpha$ for all meta-variables $\alpha \in \mathcal{V}$.

The above case distinction is complete as it covers all possible cases. Under the assumption $C_1 \doteq_L C_2$, only case 3 is possible. Hence, the implication holds.

($\Longleftarrow$) The argument is by induction on the number of meta-variables occurring in (C_1, C_2). Assume $\mathit{const}(C_1) = \mathit{const}(C_2)$ and $\forall \alpha \in \mathcal{V}: |C_1|_\alpha = |C_2|_\alpha$.

Base case: No meta-variables occur in (C_1, C_2). We obtain $C_1 \doteq_L C_2$ from $\mathit{const}(C_1) = \mathit{const}(C_2)$ and $C_1, C_2 \in \mathit{Pad}_\mathcal{V}$ (argue by induction on $\mathit{const}(C_1)$).

Step case: $n+1$ meta-variables occur in (C_1, C_2) and the proposition holds for all command pairs with n or less meta-variables. Let α' be an arbitrary variable occurring in C_1 (and, hence, also in C_2). Let σ be an arbitrary substitution that is preserving and ground for C_1 and C_2. We decompose σ into two substitutions σ_1, σ_2 such that $\sigma = \sigma_2 \circ \sigma_1$, $\mathit{dom}\,\sigma_1 = \{\alpha'\}$, and $\mathit{dom}\,\sigma_2 = \mathit{dom}\,\sigma \setminus \{\alpha'\}$. Note that σ_1 and σ_2 both are preserving. We have $|\sigma_1(C_1)|_\alpha = |C_1|_\alpha = |C_2|_\alpha = |\sigma_1(C_2)|_\alpha$ for all $\alpha \neq \alpha'$. Moreover, we have $\mathit{const}(\sigma_1(C_1)) = \mathit{const}(\sigma_1(C_2))$ because $\mathit{const}(C_1) = \mathit{const}(C_2)$ and $|C_1|_{\alpha'} = |C_2|_{\alpha'}$. From the induction assumption (n meta-variables occur in $(\sigma_1(C_1), \sigma_1(C_2))$), we obtain $\sigma_1(C_1) \doteq_L \sigma_1(C_2)$. This means, in particular, $\sigma_2(\sigma_1(C_1)) \doteq_L \sigma_2(\sigma_1(C_2))$ holds. Since $\sigma = \sigma_2 \circ \sigma_1$ and σ was chosen freely, we obtain $C_1 \doteq_L C_2$. □

A.5 Proof of Lemma 2

Lemma 2 *Let $V_i \in \boldsymbol{\mathit{Com}}$, with liftings $V_i' \in \boldsymbol{\mathit{Com}}_\mathcal{V}$ and $V_i^* \in \boldsymbol{\mathit{Mgl}}_\mathcal{V}$ for $i = 1, 2$. Suppose V_1^* (V_2^*) shares no meta-variables with V_1', V_2', and V_2^* (V_1', V_2', and V_1^*). Then we have*

$$\mathcal{U}(\{V_1' \doteq_L^? V_2'\}) \neq \emptyset \text{ implies } \mathcal{U}(\{V_1^* \doteq_L^? V_2^*\}) \neq \emptyset$$

More precisely, we can find $\rho \in \mathcal{U}(\{V_1^* \doteq_L^? V_2^*\})$ with $\mathit{dom}(\rho) \subseteq \mathit{var}(V_1^*) \cup \mathit{var}(V_2^*)$.

Proof Suppose σ is a preserving substitution with $\sigma V_1' \doteq_L \sigma V_2'$. We will inductively construct preserving substitutions ρ_1 with $\rho_1 V_1^* \doteq_L \sigma V_1'$, and ρ_2 with $\rho_2 V_2^* \doteq_L \sigma V_2'$ with the property $\mathit{dom}(\rho_i) \subseteq \mathit{var}(V_i^*)$ and $\mathit{var}(\mathit{ran}(\rho_i)) \subseteq \mathit{var}(V_i')$ for $i = 1, 2$. The meta-variables in V_1^* and V_2^* are disjoint, so $\rho = \rho_1 \cup \rho_2$ is well defined and a unifier of $V_1^* \doteq_L^? V_2^*$ because of $\rho V_1^* \doteq_L \sigma V_1' \doteq_L \sigma V_2' \doteq_L \rho V_2^*$. We prove the assertion by induction on the term structure of $V_1^* \in \boldsymbol{\mathit{Mgl}}_\mathcal{V}$, starting with $V_1^* = C_1^* \in \mathit{Mgl}_\mathcal{V}$ and hence $V_1 = C_1 \in \mathit{Com}$ and $V_1' = C_1' \in \mathit{Com}_\mathcal{V}$.

Suppose $C_1^* \in Pad_{\mathcal{V}} \cap Mgl_{\mathcal{V}}$. Then by definition of $Mgl_{\mathcal{V}}$, C_1^* contains at least one meta-variable α. C_1' is also a lifting of C_1, so it must be in $Pad_{\mathcal{V}} \cup \{\epsilon\}$. Let $\alpha_1, \ldots, \alpha_n$ be the meta-variables in C_1'. Define $\rho(\alpha) := \sigma(\alpha_1); \ldots; \sigma(\alpha_n)$, and set $\rho(Y) = \epsilon$ for all $Y \neq \alpha$ occuring in C_1^*. C_1^* and C_1' are both liftings of C_1, so they contain the same number of skips and assignments to high variables. By definition of ρ we see that $\sigma C_1'$ and ρC_1^* contain the same meta-variables and the same number of constants. Applying Lemma 1 we can conclude that $\rho C_1^* \doteq_L \sigma C_1'$. Furthermore, $dom(\rho) \subseteq var(C_1^*)$ and $var(ran(\rho)) \subseteq var(C_1')$ are satisfied.

Suppose $C_1^* = P;$ if B then $C_{1,1}^*$ else $C_{1,2}^*; C^*$. The command C_1' is also a lifting of C_1, so it can be written as $P';$ if B then $C_{1,1}'$ else $C_{1,2}'; C'$, with (possibly empty) commands P', C'.

If B is a *low* conditional, we inductively construct substitutions $\rho_1, \rho_2, \rho_3, \rho_4$ such that $\rho_1 P \doteq_L \sigma P'$, $\rho_2 C_{1,1}^* \doteq_L \sigma C_{1,1}'$, $\rho_3 C_{1,2}^* \doteq_L \sigma C_{1,2}'$ and $\rho_4 C^* \doteq_L \sigma C'$. The domains of the ρ_i are disjoint by the hypothesis that $dom(\rho_i)$ is a subset of the meta-variables of the corresponding subcommand and the assumption that every meta-variable occurs only once in C_1^*, so $\rho = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4$ is well defined. Using Lemma 9 we can conclude $\rho C_1^* = \rho_1 P;$ if B then $\rho_2 C_{1,1}^*$ else $\rho_3 C_{1,2}^*; \rho_4 C^* \doteq_L \sigma P';$ if B then $\sigma C_{1,1}'$ else $\sigma C_{1,2}'; \sigma C' \doteq_L \sigma C_1'$. Furthermore, $dom(\rho) \subseteq var(C_1^*)$ and $var(ran(\rho)) \subseteq var(C_1')$ are satisfied.

If B is a *high* conditional, the precondition $\sigma C_1' \doteq_L \sigma C_2'$ together with the definition of $\doteq_L$ on high conditionals shows that $\sigma C_{1,1}' \doteq_L \sigma C_{1,2}'$ holds. After applying the induction hypothesis, we obtain $\rho_{2,1}$ and $\rho_{2,2}$ with $\rho_{2,1} C_{1,1}^* \doteq_L \sigma C_{1,1}' \doteq_L \sigma C_{1,2}' \doteq_L \rho_{2,2} C_{1,2}^*$ and ρ_1, ρ_3 with $\rho_1 P \doteq_L \sigma P'$ and $\rho_3 C^* \doteq_L \sigma C'$. With $\rho = \rho_1 \cup \rho_{2,1} \cup \rho_{2,2} \cup \rho_3$, we see that ρC_1^* is equal to $\rho_1 P;$ if B then $\rho_{2,1} C_{1,1}^*$ else $\rho_{2,2} C_{1,2}^*; \rho_3 C^*$ and, therefore, $\rho C_1^* \doteq_L \rho_1 P;$ skip; $\rho_{2,1} C_{1,1}^*; \rho_3 C^* \doteq_L \sigma P';$ skip; $\sigma C_{1,1}'; \sigma C' \doteq_L \sigma P';$ if B then $\sigma C_{1,1}'$ else $\sigma C_{1,2}'; \sigma C' \doteq_L \sigma C_1'$. Furthermore, $dom(\rho) \subseteq var(C_1^*)$ and $var(ran(\rho)) \subseteq var(C_1')$ are satisfied.

The remaining induction steps for $Mgl_{\mathcal{V}}$ and the lifting to $\boldsymbol{Mgl_{\mathcal{V}}}$ can be treated in the same way as the low conditional case.

A.6 Proof of Lemma 3

Lemma 3 *Let $V \in \boldsymbol{Com}$ and $\overline{V} \in \boldsymbol{Com_{\mathcal{V}}}$. If $V \rightharpoonup \overline{V}$ can be derived, then*

1. *$\overline{V}$ is a lifting of V and*
2. *$\overline{V} \in \boldsymbol{Mgl_{\mathcal{V}}}$.*

Proof The proof of assertion 1 is a straightforward inductive argument over the structure of the derivation of $V \rightharpoonup \overline{V}$ and an inspection of each rule in Figure 5.

For proving assertion 2 we proceed by induction on the term structure of $V \in \boldsymbol{Com}$: First, suppose $V = C \in Com$.

If $C =$ skip, we have $C \rightharpoonup$ skip$; X$, which is in $Mgl_{\mathcal{V}}$. The same holds for $C = h := Exp$.

If $C = l := Exp$, we have $C \rightharpoonup X; l := Exp; Y$, which is in $Mgl_{\mathcal{V}}$.

If $C =$ while B do C', we get $C \rightharpoonup \overline{C} = X;$ while B do $\overline{C'}; Y$ with $C' \rightharpoonup \overline{C'}$. By induction hypothesis, $\overline{C'} \in Mgl_{\mathcal{V}}$, and by definition of $Mgl_{\mathcal{V}}$ we have $\overline{C} \in Mgl_{\mathcal{V}}$.

If $C =$ fork$(C'V)$, we have $C \rightharpoonup \overline{C} = X;$ fork$(\overline{C'V}); Y$ with $C' \rightharpoonup \overline{C'}$ and $V \rightharpoonup \overline{V}$. By induction hypothesis, $\overline{C'} \in Mgl_{\mathcal{V}}$, and $\overline{V} \in \boldsymbol{Mgl_{\mathcal{V}}}$ and by definition of $Mgl_{\mathcal{V}}$ we have $\overline{C} \in Mgl_{\mathcal{V}}$.

If $C =$ if B then C_1 else C_2, we have $C \rightharpoonup \overline{C}$ with $\overline{C} = X;$ if B then $\overline{C_1}$ else $\overline{C_2}; Y$ with $C_1 \rightharpoonup \overline{C_1}$ and $C_2 \rightharpoonup \overline{C_2}$. By induction hypothesis, $\overline{C_1}, \overline{C_2} \in Mgl_{\mathcal{V}}$, and by definition of $Mgl_{\mathcal{V}}$ we have $\overline{C} \in Mgl_{\mathcal{V}}$.

If $C = C_1; C_2$ let $C_1 \rightharpoonup \overline{C_1}$, and let $C_2 \rightharpoonup \overline{C_2}$. By induction hypothesis $\overline{C_1}, \overline{C_2} \in Mgl_{\mathcal{V}}$. As $\overline{C_1} \in Mgl_{\mathcal{V}}$, we can write it as (implicit induction on $Mgl_{\mathcal{V}}$) $\overline{C_1} = \overline{C_1'}; P; X$ for a maximal $P \in Pad_{\mathcal{V}}$ and $X \in \mathcal{V}$. Observe that $P; \overline{C_2} \in Mgl_{\mathcal{V}}$, and thus $\overline{C_1'}; P; \overline{C_2} \in Mgl_{\mathcal{V}}$. This is what we wanted, as we have $C_1; C_2 \rightharpoonup \overline{C_1'}; P; \overline{C_2}$.

If $V = \langle C_1, \ldots, C_n \rangle \in \boldsymbol{Com}$, we have $V \rightharpoonup \overline{V}$ with $\overline{V} = \langle \overline{C_1}, \ldots, \overline{C_n} \rangle$ with $C_i \rightharpoonup \overline{C_i}$ for $i = 1, \ldots, n$. By induction hypothesis, $\overline{C_1}, \ldots, \overline{C_n} \in Mgl_{\mathcal{V}}$, and thus $\overline{V} \in \boldsymbol{Mgl_{\mathcal{V}}}$.

The assertion that each meta-variable occurs at most once follows from the requirement that each meta-variable inserted while lifting must be *fresh*. □

A.7 Proof of Theorem 4

Theorem 4 *Let $V_i \in \boldsymbol{Com}$ with liftings $V_i', \overline{V_i} \in \boldsymbol{Com_{\mathcal{V}}}$ for $i = 1, 2$. Suppose $\overline{V_1}$ ($\overline{V_2}$) shares no meta-variables with V_1', V_2', and $\overline{V_2}$ (V_1', V_2', and $\overline{V_1}$). If $V_1 \rightharpoonup \overline{V_1}$ and $V_2 \rightharpoonup \overline{V_2}$ can be derived, then*

$$\mathcal{U}(\{V_1' \doteq_L^? V_2'\}) \neq \emptyset \text{ implies } \mathcal{U}(\{\overline{V_1} \doteq_L^? \overline{V_2}\}) \neq \emptyset\,.$$

Proof From Lemma 3, it follows that $\overline{V_1}, \overline{V_2} \in \boldsymbol{Mgl_{\mathcal{V}}}$. The claim then follows immediately by applying Lemma 2. □

A.8 Proof of Theorem 5

Theorem 5 *Let $V', \overline{V} \in \boldsymbol{Com_{\mathcal{V}}}$ be liftings of $V \in \boldsymbol{Com}$. Suppose $\overline{V}$ shares no meta-variables with V' and $V \rightharpoonup \overline{V}$ can be derived. Then*

$$\mathcal{U}(\{V' \doteq_L^? V'\}) \neq \emptyset \text{ implies } \mathcal{U}(\{\overline{V} \doteq_L^? \overline{V}\}) \neq \emptyset.$$

Proof From Lemma 3 follows that $\overline{V} \in \boldsymbol{Mgl_{\mathcal{V}}}$, hence it suffices to show the assertion for an arbitrary $V^* \in \boldsymbol{Mgl_{\mathcal{V}}}$. We will inductively construct a substitution ρ, with $\rho V^* \doteq^?_L \rho V^*$ and $dom(\rho) \subseteq var(V^*)$, starting with $V^* = C^* \in Mgl_{\mathcal{V}}$ and $V' = C' \in Com_{\mathcal{V}}$.

Suppose $C^* \in Pad_{\mathcal{V}}$. Then the identity is in $\mathcal{U}(\{C^* \doteq^?_L C^*\})$.

Suppose $C^* = P; l := Exp; C^*_1$. C' is also a lifting of C, so $C' = P'; l := Exp; C'_1$ with possibly empty P', C'. From $\sigma C' \doteq_L \sigma C'$, and the definition of $\doteq_L$ we know that $Exp : low$ and $\mathcal{U}(\{C'_1 \doteq^?_L C'_1\}) \neq \emptyset$. We apply induction hypothesis to obtain $\rho \in \mathcal{U}(\{C^*_1 \doteq^?_L C^*_1)$. By Lemma 9 we obtain $\rho C^* \doteq_L \rho C^*$.

Suppose $C^* = P;$ if B then C^*_1 else $C^*_2; C^*_3$ with $B : low$. C' is also a lifting of C, so $C' = P';$ if B then C'_1 else $C'_2; C'_3$. From $\sigma C' \doteq_L \sigma C'$, and the definition of $\doteq_L$ we know that $\mathcal{U}(\{C'_i \doteq^?_L C'_i\}) \neq \emptyset$ for $i = 1, 2, 3$. We apply induction hypothesis to obtain $\rho_i \in \mathcal{U}(\{C^*_i \doteq^?_L C^*_i\})$ for $i = 1, 2, 3$. $\rho = \rho_1 \cup \rho_2 \cup \rho_3$ is well-defined as the domains are pairwise disjoint, and $\rho \in \mathcal{U}(\{C^* \doteq^?_L C^*\})$.

Suppose $C^* = P;$ if B then C^*_1 else $C^*_2; C^*_3$ with $B : high$. We then know that $C' = P';$ if B then C'_1 else $C'_2; C'_3$ because both C' and C^* are liftings of C. From $\sigma C' \doteq_L \sigma C'$, and the definition of $\doteq_L$ we have $\sigma C'_1 \doteq_L \sigma C'_2$. By Lemma 2 we obtain ρ_1 with $\rho_1 C^*_1 \doteq_L \rho_1 C^*_2$. (Note that every meta-variable occurs at most once in (C^*_1, C^*_2)). Applying induction hypothesis to P and C^*_3 we obtain ρ_0 with $\rho_0 P \doteq_L \rho_0 P$ and ρ_2 with $\rho_2 C^*_3 \doteq_L \rho_2 C^*_3$. With $\rho = \rho_0 \cup \rho_1 \cup \rho_2$ we have $\rho C^* \doteq_L \rho C^*$, which is what we wanted.

The remaining induction steps for $Mgl_{\mathcal{V}}$ and the lifting to $\boldsymbol{Mgl_{\mathcal{V}}}$ can be treated in the same way as the low conditional case. □

A.9 Proof of Lemma 4

Lemma 4 *Let $V_1, V_2 \in \boldsymbol{Slice_{\mathcal{V}}}$ where no meta-variable occurs more than once in (V_1, V_2). If $V_1 \doteq^?_L V_2 :: \eta$, then*

1. *$\eta \in \mathcal{U}(\{V_1 \doteq^?_L V_2\})$, and*
2. *η is idempotent, and*
3. *$dom(\eta) \cup var(ran(\eta)) \subseteq var(V_1) \cup var(V_2)$, and*
4. *$V_1, V_2 \in \boldsymbol{Mgl_{\mathcal{V}}}$ implies $\eta V_1, \eta V_2 \in \boldsymbol{Mgl_{\mathcal{V}}}$,*

where $var(\cdot)$ returns the set of meta-variables occurring in a set of commands in $\boldsymbol{Com_{\mathcal{V}}}$.

Proof For proving assertions 1, 2, and 3 we proceed by structural induction on the derivation tree $\mathcal{D}$ of the judgment $V_1 \doteq^?_L V_2 :: \eta$. Note that assertion 2 is equivalent to $dom(\eta) \cap var(ran(\eta)) = \emptyset$.

If $\mathcal{D}$ consists of an application of the rule $[Var_1]$ we have $V_1 = \alpha$ and $V_2 = C$. The assertion follows, as α does not occur in $var(C)$ by assumption. $[Var_2]$ follows similarly.

If $\mathcal{D}$ consists of an application of the rule $[Asg]$ the assertion follows directly by definition of $\doteq_L$.

If the root of $\mathcal{D}$ is an application of rule $[Seq_1]$, we have $V_1 = \alpha; C_1$ and $V_2 = C_2$, and $C_1 \doteq^?_L C_2 :: \eta$. By hypothesis, $\eta C_1 \doteq_L \eta C_2$ holds. $\eta[\alpha \backslash \epsilon]\alpha; C_1 = \eta C_1 \doteq_L \eta C_2 \doteq_L \eta[\alpha \backslash \epsilon]C_2$, as α does not occur in C_1, C_2. $[Seq'_1]$ follows similarly.

If the root of $\mathcal{D}$ is an application of rule $[Seq_2]$, we have $V_1 = \mathsf{skip}; C_1$ and $V_2 = \mathsf{skip}; C_2$, and $C_1 \doteq^?_L C_2 :: \eta$. By hypothesis, $\eta C_1 \doteq_L \eta C_2$ holds. Then, by Lemma 9 we also have $\eta(\mathsf{skip}; C_1) \doteq_L \eta(\mathsf{skip}; C_2)$.

If the root of $\mathcal{D}$ is the application of $[Ite]$, we have $V_1 = \mathsf{if}\ B_1\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ and $V_2 = \mathsf{if}\ B_2\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2$, together with $B_1 \equiv B_2$ and $C_1 \doteq^?_L C'_1 :: \eta_1$, $C_2 \doteq^?_L C'_2 :: \eta_2$. By hypothesis we have $\eta_i \in \mathcal{U}(\{C_i \doteq^?_L C'_i\})$, and $dom(\eta_i) \cup var(ran(\eta_i)) \subseteq var(C_i) \cup var(C'_i)$ for $i = 1, 2$. As $(var(C_1) \cup var(C'_1))$ and $(var(C_2) \cup var(C'_2))$ are disjoint by hypothesis, $\eta = \eta_1 \cup \eta_2$ is well-defined and $dom(\eta) \cup var(ran(\eta)) \subseteq var(V_1) \cup var(V_2)$ and $dom(\eta) \cap var(ran(\eta)) = \emptyset$ hold. With the help of Lemma 9 we see that η is indeed a unifier. The remaining cases can be proved along the same lines.

We prove assertion 4 by induction on the term structure of V_1. We treat the case of commands $V_1 = S_1$ and $V_2 = S_2 \in Slice_{\mathcal{V}}$ first.

Suppose $S_1 \in Stut_{\mathcal{V}}$. Only the rules $[Var_1]$,$[Var_2]$, $[Seq_1]$, $[Seq'_1]$ and $[Seq_2]$ apply, so $S_2 \in Stut_{\mathcal{V}}$. From $S_1, S_2 \in Mgl_{\mathcal{V}}$ we see that both commands contain a terminal meta-variable. The two base cases for the derivation, $[Var_1]$ and $[Var_2]$ map the terminal variable at the end of one command to the end of the other command. Thus we see that both ηS_1 and ηS_2 have terminal meta-variables. As (S_1, S_2) contains every meta-variable only once, the same holds for ηS_1 and ηS_2, and hence they are elements of $Mgl_{\mathcal{V}}$.

Suppose $S_1 = P_1;$ if B_1 then $S_{1,1}$ else $S_{1,2}; S_{1,3}$ with $B : low$. From $S_1 \doteq^?_L S_2 :: \eta$ and the rules $[Ite]$ $[Seq_3]$ and $[Seq_4]$ in Figure 6 we get $S_2 = P_2;$ if B_2 then $S_{2,1}$ else $S_{2,2}$; $S_{2,3}$, with $B_1 \equiv B_2$ and $P_1 \doteq^?_L P_2 :: \eta_0$, $S_{1,1} \doteq^?_L S_{2,1} :: \eta_1$, $S_{1,2} \doteq^?_L S_{2,2} :: \eta_2$, and $S_{1,3} \doteq^?_L S_{2,3} :: \eta_3$ are derivable. By induction hypothesis have $\eta_0 P_1$, $\eta_1 S_{1,1}$, $\eta_2 S_{1,2}$, $\eta_3 S_{1,3} \in Mgl_{\mathcal{V}}$. With $\eta = \eta_0 \cup \eta_1 \cup \eta_2 \cup \eta_3$ and the fact that the domains and variable ranges of the η_i are mutually disjoint we have $\eta S_1 = \eta_0 P_1;$ if B_1 then $\eta_1 S_{1,1}$ else $\eta_2 S_{1,2}$; $\eta_3 S_{1,3}$, which is in $Mgl_{\mathcal{V}}$ as it contains every meta-variable at most once.

All other constructors and the lifting to command vectors can be treated in the same way as the low conditional. □

Fig. 7 Resembling commands

$$\frac{P, P' \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\}}{P \doteq P'} \qquad \frac{P, P' \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\} \quad C \doteq C' \quad Id : low \quad Exp_1 \equiv Exp_2}{P; Id{:=}Exp_1; C \doteq P'; Id{:=}Exp_2; C'}$$

$$\frac{P, P' \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\} \quad C_i \doteq C_i' \quad i = 1,2,3 \quad B_1 \equiv B_2}{P; \mathsf{if}\ B_1\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3 \doteq P'; \mathsf{if}\ B_2\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'; C_3'} \qquad \frac{P, P' \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\} \quad C_1 \doteq C_1' \quad C_2 \doteq C_2' \quad B_1 \equiv B_2}{P; \mathsf{while}\ B_1\ \mathsf{do}\ C_1; C_2 \doteq P'; \mathsf{while}\ B_2\ \mathsf{do}\ C_1'; C_2'}$$

$$\frac{P, P' \in \mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\} \quad C_1 \doteq C_1' \quad C_2 \doteq C_2' \quad V \doteq V'}{P; \mathsf{fork}(C_1 V); C_2 \doteq P'; \mathsf{fork}(C_1' V'); C_2'} \qquad \frac{C_1 \doteq C_1', \ldots, C_n \doteq C_n'}{\langle C_1, \ldots, C_n \rangle \doteq \langle C_1', \ldots, C_n' \rangle}$$

A.10 Proof of Theorem 6

Theorem 6 *Let $V \in \boldsymbol{Com}$, $\overline{V}, W \in \boldsymbol{Com}_{\boldsymbol{\mathcal{V}}}$, W be a lifting of V, and $V \rightharpoonup \overline{V}$.*

1. *If there is a preserving substitution σ with $\sigma W \doteq_L \sigma W$, then $\overline{V} \hookrightarrow' V' : S$ for some $V', S \in \boldsymbol{Com}_{\boldsymbol{\mathcal{V}}}$.*
2. *If $W \hookrightarrow W' : S$ for some $W', S \in \boldsymbol{Com}_{\boldsymbol{\mathcal{V}}}$ then $\overline{V} \hookrightarrow' V' : S'$ for some $V', S' \in \boldsymbol{Com}_{\boldsymbol{\mathcal{V}}}$.*

The proof of Theorem 6 essentially proceeds by induction on the term structure of W. The main technical difficulty lies in showing that if the two branches C_1, C_2 of a conditional with high guard, with $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2$, unify, then we can also unify the corresponding slices, i.e., $S_1 \doteq_L^? S_2 :: \eta$. To prove this, we proceed in two steps:

1. We show that the slices corresponding to two commands with unifier are structurally equivalent "modulo" commands in $\mathit{Pad}_{\mathcal{V}}$, and that they are in $\mathit{Mgl}_{\mathcal{V}}$ (Definition 12, Lemma 10).
2. We show that we can unify every two structurally equivalent slices, given that they are elements of $\mathit{Mgl}_{\mathcal{V}}$ (Lemma 12.1)

Lemma 11 and Lemma 12.2 are rather technical and will be needed during the proof. We first formulate the above steps in terms of definitions and lemmata, before we proceed with the proof of Theorem 6.

To simplify the atomic treatment of subcommands in $\mathit{Stut}_{\mathcal{V}}$ of commands in $\mathit{Slice}_{\mathcal{V}}$ in inductive arguments, we introduce the language $\mathit{Slice}^+_{\mathcal{V}}$. (Note the resemblance to the definition of $\mathit{Mgl}_{\mathcal{V}}$). Define the set $\mathit{Slice}^+_{\mathcal{V}}$ by the grammar:

$$L ::= P \mid P; Id_l := Exp; L \mid P; \mathsf{if}\ B\ \mathsf{then}\ L_1\ \mathsf{else}\ L_2; L \mid P; \mathsf{while}\ B\ \mathsf{do}\ L_1; L \mid P; \mathsf{fork}(L_1 V); L$$

where L, L_1, L_2 are placeholders for commands in $\mathit{Slice}^+_{\mathcal{V}}$, V is a placeholder for a command vector in $\boldsymbol{Slice}^+_{\boldsymbol{\mathcal{V}}}$, and P is a placeholder for a command in $\mathit{Stut}_{\mathcal{V}} \cup \{\epsilon\}$. By a straightforward induction one proves that $\mathit{Slice}_{\mathcal{V}} \subseteq \mathit{Slice}^+_{\mathcal{V}}$.

Definition 12 The binary relation $\doteq$ on $\boldsymbol{Slice}^+_{\boldsymbol{\mathcal{V}}}$ is defined as the reflexive, symmetric and transitive closure of the relation inductively defined in Fig. 7. We call commands $V, V' \in \boldsymbol{Mgl}_{\boldsymbol{\mathcal{V}}}$ with $V \doteq V'$ *resembling*.

Lemma 10 *Let $C, C_1, C_2 \in \mathit{Mgl}_{\mathcal{V}}$. Then the following assertions hold:*

1. *$C \hookrightarrow' C' : S$ implies $S \in \mathit{Mgl}_{\mathcal{V}} \cap \mathit{Slice}_{\mathcal{V}}$ and $\mathit{var}(S) \subseteq \mathit{var}(C)$.*
2. *$\mathcal{U}(\{C_1 \doteq_L^? C_2\}) \neq \emptyset$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2$ implies $S_1 \doteq S_2$.*

Proof 1. It is easy to see that $S \in \mathit{Slice}_{\mathcal{V}}$ holds, so we concentrate on containment in $\mathit{Mgl}_{\mathcal{V}}$. We proceed by induction on the term structure of $C \in \mathit{Mgl}_{\mathcal{V}}$.
Suppose $C \in \mathit{Pad}_{\mathcal{V}} \cap \mathit{Mgl}_{\mathcal{V}}$. By definition of $\mathit{Mgl}_{\mathcal{V}}$, C has a terminal meta-variable. Then clearly $S \in \mathit{Mgl}_{\mathcal{V}}$ as it contains a terminal meta-variable and no assignments. The condition on the meta-variables is fulfilled.
Suppose now $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : \mathit{low}$ and $C \hookrightarrow' C' : S$. Then by definition of $\hookrightarrow'$ we have $P \hookrightarrow' P' : S_0$, $C_1 \hookrightarrow' C_1' : S_1$, $C_2 \hookrightarrow' C_2' : S_2$ and $C_3 \hookrightarrow' C_3' : S_3$. By induction hypothesis, $S_0, S_1, S_2, S_3 \in \mathit{Mgl}_{\mathcal{V}}$ and the condition on the meta-variables holds. By definition of $\mathit{Mgl}_{\mathcal{V}}$, $S = S_0; \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; S_3 \in \mathit{Mgl}_{\mathcal{V}}$, and the condition on the meta-variables follows.
Suppose now $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : \mathit{high}$. We have $C \hookrightarrow' C' : S$, so by definition of $\hookrightarrow'$ we have $P \hookrightarrow' P' : S_0$, $C_1 \hookrightarrow' C_1' : S_1$, $C_2 \hookrightarrow' C_2' : S_2$ and $C_3 \hookrightarrow' C_3' : S_3$ and also $S_1 \doteq_L S_2 :: \eta$ for some η. By induction hypothesis, $S_1, S_2, S_3 \in \mathit{Mgl}_{\mathcal{V}}$ and the condition on the meta-variables holds. As $\mathit{var}(S_i) \subseteq \mathit{var}(C_i)$, (S_1, S_2) contains every meta-variable at most once. With Lemma 4.4 we see that $\eta S_1 \in \mathit{Mgl}_{\mathcal{V}}$, and every meta-variable occurs at most once in ηS_1. From Lemma 4.3 it follows that $\mathit{dom}(\eta) \cup \mathit{ran}(\mathit{var}(\eta))$ is a subset of the meta-variables in S_1 and S_2 and hence the condition on the meta-variables is fulfilled for $S = (S_0; \mathsf{skip}; \eta S_1); S_3$. $S_0; \mathsf{skip} \in \mathit{Stut}_{\mathcal{V}}$, and so by the definition of $\mathit{Mgl}_{\mathcal{V}}$, $S_0; \mathsf{skip}; \eta S_1 \in \mathit{Mgl}_{\mathcal{V}}$. By a straightforward induction one shows that $D_1; D_2 \in \mathit{Mgl}_{\mathcal{V}}$ whenever $D_1, D_2 \in \mathit{Mgl}_{\mathcal{V}}$, and we see that $S = (S_0; \mathsf{skip}; \eta S_1); S_3$ and hence in $\mathit{Mgl}_{\mathcal{V}}$.

The cases for the other constructors follow along the same lines as the low conditional.

2. Let $\sigma C_1 \doteq_L \sigma C_2$. By symmetry and transitivity of $\doteq_L$ we conclude $\sigma C_1 \doteq_L \sigma C_1$ and $\sigma C_2 \doteq_L \sigma C_2$. With help of Lemma 11 we obtain $C_1'', C_2'' \in \mathit{Slice}_\mathcal{V}$ with $S_1 \doteq C_1'' \doteq_L \sigma C_1 \doteq_L \sigma C_2 \doteq_L C_2'' \doteq S_2$. Lemma 12.1 shows that $C_1'' \doteq_L C_2''$ implies $C_1'' \doteq C_2''$, and by transitivity of $\doteq$ we get $S_1 \doteq S_2$. □

Lemma 11 *Let $C \in \mathit{Mgl}_\mathcal{V}$ with $\sigma \in \mathcal{U}(\{C \doteq_L C\})$ and $C \hookrightarrow' C' : S$. Then there is $C'' \in \mathit{Slice}_\mathcal{V}$ with $C'' \doteq_L \sigma C$ and $C'' \doteq S$.*

Proof We proceed by structural induction on $C \in \mathit{Mgl}_\mathcal{V}$.

Suppose $C \in \mathit{Pad}_\mathcal{V}$. Choose C'' as σC, where all assignments to high variables are replaced by skips. We have $C'' \in \mathit{Stut}_\mathcal{V}$. We also have $S \in \mathit{Stut}_\mathcal{V}$, and so $C'' \doteq S$.

Suppose $C = P;\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : \mathit{low}$. We have $\sigma C \doteq_L \sigma C$, so by definition of $\doteq_L$ we obtain $\sigma P \doteq_L \sigma P$ and $\sigma C_i \doteq_L \sigma C_i$ for $i = 1,2,3$. From the precondition $C \hookrightarrow' C' : S$ and the definition of $\hookrightarrow'$ we get $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2,3$. We apply induction hypothesis to the corresponding command-pairs and obtain P'', C_i'' with $P'' \doteq_L \sigma P$, $P'' \doteq S_0$, and $C_i'' \doteq_L \sigma C_i$, $C_i'' \doteq S_i$ for $i = 1,2,3$. Then we can conclude $C'' = P'';\mathsf{if}\ B\ \mathsf{then}\ C_1''\ \mathsf{else}\ C_2''; C_3'' \doteq_L \sigma C$, as well as $C'' \doteq S$.

Suppose $C = P;\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : \mathit{high}$. We have $\sigma C \doteq_L \sigma C$, so by definition of $\doteq_L$ we obtain $\sigma P \doteq_L \sigma P, \sigma C_1 \doteq_L \sigma C_1 \sigma C_3 \doteq_L \sigma C_3$ and $\sigma C_1 \doteq_L \sigma C_2$. By symmetry and transitivity of $\doteq_L$ we conclude $\sigma C_2 \doteq_L \sigma C_2$. From the precondition $C \hookrightarrow' C' : S$ and the definition of $\hookrightarrow'$ we get $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2,3$. We apply induction hypothesis to the corresponding command-pairs and obtain P'', C_i'' with the desired properties for $i = 1,2,3$. Define C'' as $P'';\mathsf{skip}; C_1''; C_3'' \in \mathit{Slice}_\mathcal{V}$. We have $C'' \doteq_L \sigma C$ by Lemma 9. On the other hand, $S = S_0;\mathsf{skip};\eta S_1; S_3$ for some preserving substitution η. We are left to show $S \doteq C''$. From hypothesis we know $C_1'' \doteq S_1$ and $C_3'' \doteq S_3$. A straightforward induction shows that $S_1 \doteq \eta S_1$ for every preserving η. Another straightforward induction shows that $\doteq$ is a congruence with respect to sequential composition and so we conclude $C'' = P'';\mathsf{skip}; C_1''; C_3'' \doteq S_0;\mathsf{skip};\eta S_1; S_3 = S$.

The other constructors follow along the same lines as the low conditional. □

Lemma 12 *Let $S_1, S_2 \in \mathit{Slice}_\mathcal{V}$. Then the following holds:*

1. *$S_1 \doteq_L S_2$ implies $S_1 \doteq S_2$.*
2. *if $S_1, S_2 \in \mathit{Mgl}_\mathcal{V}$ and $S_1 \doteq S_2$ then there is an η with $S_1 \doteq^?_L S_2 :: \eta$.*

Proof We prove assertion 1 by induction on the structure of S_1, where we make use of the fact that $\mathit{Slice}_\mathcal{V} \subseteq \mathit{Slice}^+_\mathcal{V}$. If $S_1 \in \mathit{Stut}_\mathcal{V}$, then by definition of $\doteq_L$ and the precondition $S_2 \in \mathit{Slice}_\mathcal{V}$ we know that $S_2 \in \mathit{Stut}_\mathcal{V}$, and hence $S_1 \doteq S_2$. If $S_1 = P_1;\mathsf{if}\ B_1\ \mathsf{then}\ S_{1,1}\ \mathsf{else}\ S_{1,2}; S_{1,3}$ with $B_1 : \mathit{low}$, then by definition of $\doteq_L$ we know that $S_2 = P_2;\mathsf{if}\ B_2\ \mathsf{then}\ S_{2,1}\ \mathsf{else}\ S_{2,2}; S_{2,3}$ with $P_1, P_2 = \epsilon$ or $P_1 \doteq_L P_2$, and $S_{1,i} \doteq_L S_{2,i}$ for $i = 1,2,3$ and $B_1 \equiv B_2$. By definition of $\doteq$, $P_1 \doteq P_2$ holds, and by induction hypothesis we see $S_{1,i} \doteq S_{2,i}$ for $i = 1,2,3$. By definition of $\doteq$ we conclude $S_1 \doteq S_2$. The other constructors follow in a similar fashion.

We prove assertion 2 by induction on the term structure of S_1:

Suppose $S_1 \in \mathit{Stut}_\mathcal{V}$. Then by definition of $\doteq$, S_2 must also be in $\mathit{Stut}_\mathcal{V}$. $S_1, S_2 \in \mathit{Mgl}_\mathcal{V}$, hence they have terminal meta-variables. A simple induction on the length of S_1 shows that $S_1 \doteq_L S_2 :: \eta$ is derivable.

Suppose $S_1 = P_1;\mathsf{if}\ B_1\ \mathsf{then}\ S_{1,1}\ \mathsf{else}\ S_{1,2}; S_{1,3}$, where $P_1, S_{1,1}, S_{1,2}, S_{1,3}$ are elements of $\mathit{Mgl}_\mathcal{V} \cap \mathit{Slice}_\mathcal{V}$. By definition of $\doteq$ and $\mathit{Mgl}_\mathcal{V}$ we get $S_2 = P_2;\mathsf{if}\ B_2\ \mathsf{then}\ S_{2,1}\ \mathsf{else}\ S_{2,2}; S_{2,3}$ with $P_2, S_{2,1}, S_{2,2}, S_{2,3} \in \mathit{Mgl}_\mathcal{V} \cap \mathit{Slice}_\mathcal{V}$ and $P_1 \doteq P_2$, $S_{1,i} \doteq S_{2,i}$ for $i = 1,2,3$, and $B_1 \equiv B_2$. We apply induction hypothesis and obtain $P_1 \doteq^?_L P_2 :: \sigma_0$, and $S_{1,i} \doteq^?_L S_{2,i} :: \sigma_i$ for $i = 1,2,3$. We can conclude $S_1 \doteq^?_L S_2 :: \sigma$ with $\sigma = \bigcup_{i=0}^3 \sigma_i$ by definition of the unification calculus. The other cases follow along the same lines. □

Proof of Theorem 6:

1. Let W be an arbitrary lifting of V. From Theorem 5 follows that $\sigma W \doteq_L \sigma W$ implies $\mathcal{U}(\overline{V} \doteq_L \overline{V}) \neq \emptyset$. From Lemma 3 we see that $\overline{V} \in \mathit{Mgl}_\mathcal{V}$.
   Restricting ourselves to commands in $\mathit{Com}_\mathcal{V}$ for the moment and substituting C for $\overline{V}$, it suffices to show the assertion

   $$\exists \sigma . \sigma C \doteq_L \sigma C \Rightarrow C \hookrightarrow' C' : S$$

   for all $C \in \mathit{Mgl}_\mathcal{V}$. We proceed by induction on the term structure of C.
   If $C \in \mathit{Pad}_\mathcal{V}$, we always have $C \hookrightarrow' C' : S$.
   If $C = P; \mathit{Id}_l := \mathit{Exp}; C_1$, and $\sigma C \doteq_L \sigma C$, then we have $\sigma P \doteq_L \sigma P$ and $\sigma C_1 \doteq_L \sigma C_1$ and $\mathit{Exp} : \mathit{low}$ by definition of $\doteq_L$. By applying induction hypothesis we obtain $P \hookrightarrow' P' : S_0$ and $C_1 \hookrightarrow' C_1' : S_1$. By definition of $\hookrightarrow'$ this implies $C \hookrightarrow' P'; \mathit{Id}_l := \mathit{Exp}; C_1' : S_0; \mathit{Id}_l := \mathit{Exp}; S_1$.
   If $C = P;\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : \mathit{low}$, and $\sigma C \doteq_L \sigma C$, then we have $\sigma P \doteq_L \sigma P$ and $\sigma C_i \doteq_L \sigma C_i$ for $i = 1,2,3$. By applying induction hypothesis we obtain $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2,3$. By definition of $\hookrightarrow'$ this implies $C \hookrightarrow' P';\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'; C_3' : S_0;\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; S_3$

If $C = P;\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : high$, and $\sigma C \doteq_L \sigma C$, then we have $\sigma P \doteq_L \sigma P$ and $\sigma C_i \doteq_L \sigma C_i$ for $i = 1, 3$. Furthermore we have $\sigma C_1 \doteq_L \sigma C_2$, from which we get $\sigma C_2 \doteq_L \sigma C_2$ by transitivity and symmetry of $\doteq_L$. Induction hypothesis yields $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1, 2, 3$. Every meta-variable occurs at most once in C, hence the same holds true for the subterms C_1, C_2.
Lemma 10 shows that $S_1, S_2 \in Mgl_{\mathcal{V}} \cap Slice_{\mathcal{V}}$, $S_1 \doteq S_2$, and every meta-variable occurs at most once in (S_1, S_2). Lemma 12.2 implies that there is η with $S_1 \doteq^?_L S_2 :: \eta$. η is a unifier of S_1, S_2, as Lemma 4 shows. We conclude $C \hookrightarrow' P'; \mathsf{if}\ B\ \mathsf{then}\ \eta C_1'\ \mathsf{else}\ \eta C_2'; C_3' : S_0; \mathsf{skip}; \eta S_1; S_3$, which is what we wanted. The cases for the other constructors follow along the same lines as the low conditional. The assertion can then simply be lifted to command vectors.

2. By a straightforward induction over the derivation tree it follows that $W \hookrightarrow W' : S$ implies $W' = \sigma W$ for a preserving substitution σ. In the proof of Theorem 3 it was shown that $S \doteq_L W'$ holds. By symmetry and transitivity of $\doteq_L$ we obtain $\sigma W \doteq_L \sigma W$. The assertion now follows directly from part 1 of Theorem 6. □

References

1. Agat, J.: Transforming out timing leaks. In: Proceedings of the 27th ACM Symposium on Principles of Programming Languages, pp. 40–53 (2000)
2. Barthe, G., Rezk, T., Warnier, M.: Preventing timing leaks through transactional branching instructions. In: Proceedings of 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05). ENTCS, (2005)
3. Baader, F., Snyder, W.: Unification theory. In: Robinson, A., Voronkov, A. (eds.), Handbook of Automated Reasoning, vol. I, chap. 8, pp. 445–532. Elsevier Science, Amsterdam (2001)
4. Dam, M.: Decidability and proof systems for language-based noninterference relations. In: Proceedings of the 33rd ACM Symposium on Principles of Programming Languages, pp. 67–78 (2006)
5. Denning, D.E.: Cryptography and Data Security. Addison-Wesley, New York (1982)
6. Hedin, D., Sands, D.: Timing aware information flow security for a javacard-like bytecode. Electr. Notes Theor. Comput. Sci. **141**(1), 163–182 (2005)
7. Herold, A., Siekmann, J.: Unification in Abelian semi-groups. J. Autom. Reason. **3**, 247–283 (1987)
8. Köpf, B., Mantel, H.: Eliminating implicit information leaks by transformational typing and unification. In: Proceedings of FAST'05: 3rd International Workshop on Formal Aspects in Security and Trust. LNCS, vol. 3866, pp. 47–62. Springer, Heidelberg (2006)
9. Mantel, H., Sands, D.: Controlled declassification based on intransitive noninterference. In: Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004. LNCS, vol. 3303, pp. 129–145, Taipei, Taiwan, 4–6 November 2004. Springer, Heidelberg (2004)
10. McLean, J.D.: A general theory of composition for trace sets closed under selective interleaving functions. In: Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 79–93, Oakland, CA, USA (1994)
11. Sabelfeld, A.: The impact of synchronisation on secure information flow in concurrent programs. In: Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics. vol. 2244, pp. 225–239 (2001)
12. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communication **21**(1), 5–19 (2003)
13. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. In: Proceedings of the 8th European Symposium on Programming, LNCS, pp. 50–59 (1999)
14. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings of the 13th IEEE Computer Security Foundations Workshop, pp. 200–215, Cambridge, UK (2000)
15. Schneider, B.F.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
16. Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. In: Proceedings of the 11th IEEE Computer Security Foundations Workshop, pp. 34–43, Rockport, Massachusetts (1998)